University of Oslo
Department of Informatics

Design, Implementation, and Evaluation of Network Monitoring Tasks with the TelegraphCQ Data Stream Management System

Master's Thesis

Jarle Søberg

1st May 2006

Preface

This Master's Thesis is written at the Distributed Multimedia Systems Research Group at the Department of Informatics, University of Oslo, between August 2004 to May 2006.

I want to thank my supervisors, Vera Goebel and Thomas Plagemann for their support, ideas, and guidance. I would also like to thank Ovidiu Valentin Drugan for discussing everything from network analysis and TelegraphCQ, to backup routines. Martin Normann Nielsen and Marius Frøisland have read through the thesis and commented the content, and Dag Ivan Homlong has assisted me with my English. The TelegraphCQ mailing list has been a great help, as well.

Special thanks go to Kjetil Helge Hernes for great teamwork and cooperation throughout this project.

Jarle Søberg
University of Oslo
May, 2006

Abstract

Data stream management systems (DSMSs) provide a new and alternative way to perceive and analyze data streams. Similar to the database management systems (DBMSs), the DSMSs use a declarative query language to handle data. One of the main differences is that the DSMSs obtain the data from streaming sources, for example a local area network (LAN), instead of a database. Such an approach opens up for a set of tasks that can be described using e.g. SQL-like queries.

To begin with, we discuss the networking application and introduce a set of requirements that might be useful for DSMSs in general. Some of these requirements are further discussed as we describe the issues in DSMSs. This thesis focuses on one particular DSMS, TelegraphCQ, and we give a thorough description and discussion of its features.

We have designed and implemented a set of tasks that may be of value for the network monitoring application as described in this thesis. We discuss these tasks, investigate their qualities and propose solutions on how to implement them in the declarative language provided by TelegraphCQ.

Finally, we run a performance analysis of some of the tasks to see how TelegraphCQ manages to handle data streams at varying loads. We focus on two metrics; relative throughput to the number of packets received, and accuracy of the results. These metrics are very important with respect to the reliability and applicability of TelegraphCQ. In this context, we implement an experiment setup for network monitoring with DSMSs, such that the results can be easily re-tested and verified. We show that TelegraphCQ only manages a network load of approximately 2.5 Mbits/s before it starts dropping packets.

We end the discussion by evaluating TelegraphCQ's support for the requirements described in the beginning of the thesis, and point out some of the requirements TelegraphCQ does not support. We discuss the results from the performance evaluation and conclude that the accuracy is satisfying. The conclusion is that, due to the low relative throughput, TelegraphCQ is not suited for network traffic monitoring at higher network loads.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

The Internet is considered the most influential factor in today's information based society. Thus, it is important to monitor the Internet and its data traffic. By monitoring, measuring, and analyzing the data that is sent both inside and between the thousands of networks the Internet consists of, one gets a clearer view of patterns of usage, possible congestion, network abnormalities, and bandwidth utilization, for example.

Network monitors incorporate many important applications that need efficient and fast assessment of such data. The networks and the network protocols may be complex, as well as network traffic tends to have unpredictable behavior. Hence, it is challenging to anticipate what characteristics the network may have the next 24 hours, for instance. Network traffic in general can be considered to consist of continuous and infinite data streams of packets, and each packet contains up to several different protocol headers. The data streams do not have a well defined length or size, and may have an unpredictable behavior with respect to characteristics such as data rates and protocols.

In a network monitoring scenario, these streams can be persistently stored on a hard disk, something which may require a vast amount of storage space if the traffic load is high, or the monitoring is long lasting. When the data streams are stored, they can be analyzed and investigated. Though, this approach may not always satisfy the needs of for example reacting quickly to sudden changes in the data stream, such as congestion or network attacks.

As an alternative to the persistent storing, the data streams can be monitored in an *on-line* fashion, meaning that the packets are analyzed when they arrive, and that they are normally not stored to disk. On-line monitoring of data streams is thus considered a very challenging task, as the monitoring system has to e.g. handle protocol states deduced from the streams and analyze as many packet headers as

possible to get a satisfying overview. In order to get detailed and accurate information as well, the header fields, which the headers consists of, have to be analyzed.

Hence, such requirements demand solutions that often can be cumbersome or proprietary. The cumbersome solutions are high level language monitoring programs, written in e.g. Perl or Java, and that tend not to be informally documented, or written in such a way that even not the persons who have written them understand what the code does. Sometimes, as new functionality is needed, even more code is added. These languages, which are referred to as *procedural* languages [pro], focus on *how*, i.e., which procedures to use, to solve the tasks.

There are also a number of available, commercial network monitoring tools, but as these tend to be proprietary, they give none, or few, possibilities for investigating with regard to efficiency of certain components. For example, the proprietary solutions might also have a strict interface, posing no alternative solutions for ways of monitoring data traffic. The parameters mentioned above might make it hard to develop new solutions or confirm other's results.

Many database management systems (DBMSs) use the *declarative* query language, SQL, which is applied to fetch and operate on persistent tuples from a given set of databases. Declarative languages are generally used to tell *what* to do, instead of *how* to do it [dec]. SQL, which is an open standard [EM99], offers a syntax that makes it intuitive to understand, and easy to learn. Due to this strictness, the queries are understandable for most people with programming experience. By offering this simplicity, performance evaluations, discussions, and e.g. creating of workbenches have a common platform of understanding. This is believed to have a positive effect on the overall bi-lateral evaluation of experimental results.

Recent research has investigated the possibility of using the ideas posed in the declarative database query languages to use these features in systems that, instead of explicitly obtaining data from a database, receive the data as a stream *pushed* to the system. These systems make use of SQL's possibilities of for example aggregations and joins, just as if the data source was a database. These systems, the *data stream management systems* (DSMSs), are developed to propose a solution for using SQL-like declarative syntax to get information from data streams for stream analysis purposes. In these DSMSs, the data packet is viewed as a tuple, and the header fields can be viewed as attributes. The DSMSs are promising, as they pose a new and adequate way of investigating data streams efficiently.

As we see throughout this thesis, solving on-line monitoring challenges by introducing declarative languages and DSMSs is not as simple as it might seem in the first place. One of the main challenges is e.g. how to implement joins, which are blocking, on a data stream that is considered infinite. The literature and existing implementations propose suggestions for solving these challenges, as well as others. Another challenge is how to pose the language so that it efficiently and intuitively provides the set of functions needed for handling streaming applications.

Though, it is important to know the basics of DSMSs. This makes it simpler to

understand why the DSMS technology poses such a new way of understanding how to work with on-line data stream monitoring. It is also important to know whether or not certain DSMSs can handle general and special network monitoring tasks, and how well they manage to solve them.

1.2 Problem Description

There are several DSMSs available through the great number of projects investigating this technology [ACC+03, ZSC+03, CDTW00, ABB+04, CCD+03, CJSS03]. Thus, there are several implementations that are public domain and available for downloading, further developing, and testing. On a general basis, the DSMSs are far from perfect, as they are mainly developed for research purpose and e.g. only investigate some features. Though, the developers and researchers are still interested in feedback from users, to further add features, and to change the DSMSs such that they provide a simple and powerful user interface. This motivates for even more thoroughly testing and researching.

We have chosen to focus on one particular DSMS; TelegraphCQ, which is developed by the UC Berkeley's Computer Science Division. As part of a project at Institut Eurécom [Eur], Plagemann et al. [PGB+04] used an earlier version of TelegraphCQ to perform a set of network traffic monitoring tasks. The article concludes that TelegraphCQ is not suitable as a general tool network traffic monitoring. They base the conclusion on certain limitations in the version of TelegraphCQ they were using. The newest version of TelegraphCQ supports some of the functionality that was missed in the prior version. Thus, a follow-up evaluation is needed in order to see if it is possible to express a new and more complex set of network monitoring tasks. It is also important to evaluate TelegraphCQ's *performance* with respect to these tasks.

Hence, our field of interest is network monitoring in particular, and we try to utilize TelegraphCQ in this application domain. Thus, our problem description is:

> *Evaluate the performance of TelegraphCQ in an on-line network monitoring scenario.*

This problem description opens for several interpretations, so we specify the statement in the following.

To understand both network monitoring and how TelegraphCQ can be used in such a scenario, we build a general understanding of this by introducing application domains for data streams, issues in DSMSs, and details about TelegraphCQ. In the discussion of the data stream application domains, we introduce a set of requirements these applications generate. These requirements are important, as they are referred to throughout the thesis. We discuss some of them when we introduce

the DSMS issues. We base some of the assessments and conclusions about TelegraphCQ with respect to these requirements, as well.

By *performance of TelegraphCQ* we mean how well TelegraphCQ performs with respect to a set of metrics that we define. These metrics mainly investigate the data rate TelegraphCQ manages to handle, as well as its accuracy. The metrics reveal important qualities that indicate how reliable TelegraphCQ is in the role as a network monitoring tool. If it gives inaccurate results on input data that has certain characteristics, there is a possibility that TelegraphCQ gives inaccurate results on other data streams, as well.

Hence, we design a set of tasks that can be considered relevant for some network applications. Some of these tasks are taken from [PGB+04]. For the remaining tasks, we investigate whether they can be expressed using TelegraphCQ's query language. If they do, we perform a simple accuracy test on the query in order to investigate if the query gives correct results. This accuracy test may consist of hand-coded data packets that are sent to TelegraphCQ, or other data generating activity like for example Web browsing. We specify this when discussing the tasks. If we do not manage to implement the tasks in such a way that they can be solved using TelegraphCQ, we discuss this, and try to locate what kind of functionality we miss. This functionality is compared to the requirements mentioned in the first chapters.

By *on-line network monitoring scenario* we mean a controlled environment that sends data traffic to TelegraphCQ. The environment has characteristics representative for a real-life on-line data stream in a network. We try to keep as many of the data packet factors (e.g. protocol and header size) constant, so that only the rate in which the data is sent and received vary. We also argue that the scenario we create is correct and performs the tasks as intended. Possible aberrations from its correctness are mentioned and discussed. In this evaluation, TelegraphCQ performs *passive* network measuring, i.e., it passively listens to the traffic on the network.

TelegraphCQ is designed in a very special and adaptive way with respect to query optimization. The optimization is considered very dynamic and aim to change the internal structure depending on the stream characteristics. The scenario we create is thus a worst-case scenario with respect to the way TelegraphCQ adapts to streams. Though, we do not investigate much of TelegraphCQ's internal processes; we base our investigation on the data we send to the DSMS and the results it outputs, like a black-box. We still use some of TelegraphCQ's own introspective monitoring features that give some information about queues and operators used in the query processing, but, as introspection may not always be a correct way of investigating systems, we do not rely on those results unless they are verified empirically.

At the same time as we investigate TelegraphCQ, we also monitor the system TelegraphCQ is installed on, e.g. the CPU and memory utilization as well as number of packets entering and leaving the system. We use these data to reveal how TelegraphCQ affects the system, and to find out more about how accurate the DSMS

is.

As mentioned, during the design of the tasks, we expect that some of the tasks can not be solved correctly due to TelegraphCQ's query language definitions. We try to point out these limitations based on our understanding of how TelegraphCQ is designed. We also suggest possible improvements, such that the tasks could have been solved correctly.

1.3 Outline

The chapters in this thesis focus on different aspects of the evaluation process. Following is a short description for each of the chapters.

This project has been a collaboration between two master students focusing on two different DSMSs. We have shared much knowledge and have used the same experiment setup for our experiments. Chapter 2 is mostly written by the other master student, Kjetil Helge Hernes [Her06]. The chapter is only slightly changed to fit this thesis' structure. Chapter 2 introduces a broad view of some of the applications in which data streams are involved. The chapter investigates, e.g. the network monitoring domain, which is focused on in this thesis. Chapter 2 plays a role for introducing the general challenges in network monitoring, as well. Finally, the chapter shows some of the requirements the streaming applications make.

The theoretical background for DSMSs is introduced in Chapter 3, where the most important issues and concepts in this technology are introduced and discussed. This chapter also appears in [Her06].

TelegraphCQ is thoroughly discussed in Chapter 4, where some of the most important features are shown. The chapter also discusses how TelegraphCQ is designed, and how it implements some the concepts described in Chapter 3. The final part of the chapter gives a practical overview of TelegraphCQ.

Since the first three chapters give the theoretical background for DSMSs, a reader that already has detailed knowledge of application domains for data streams, DSMSs, and TelegraphCQ may not need to to read these chapters in detail. However, it is recommended to read these chapters, since the performance evaluation relies on the theoretical background they provide. In addition, at the time of writing, Chapter 4 is probably the most extensive introduction to TelegraphCQ available, as it focuses on both theoretical and practical issues.

The set of tasks are designed and implemented in Chapter 5. We discuss each of the tasks and aim to show which DSMS requirements they investigate. We also show some tasks that are not possible to implement given TelegraphCQ's query language.

Chapter 6 describes and discusses how the *experiment setup* is implemented. In Chapter 6, the experiment setup is defined to be the computers used in the exper-

iments, the monitoring tools used, i.e., everything used in the evaluation *except* TelegraphCQ.

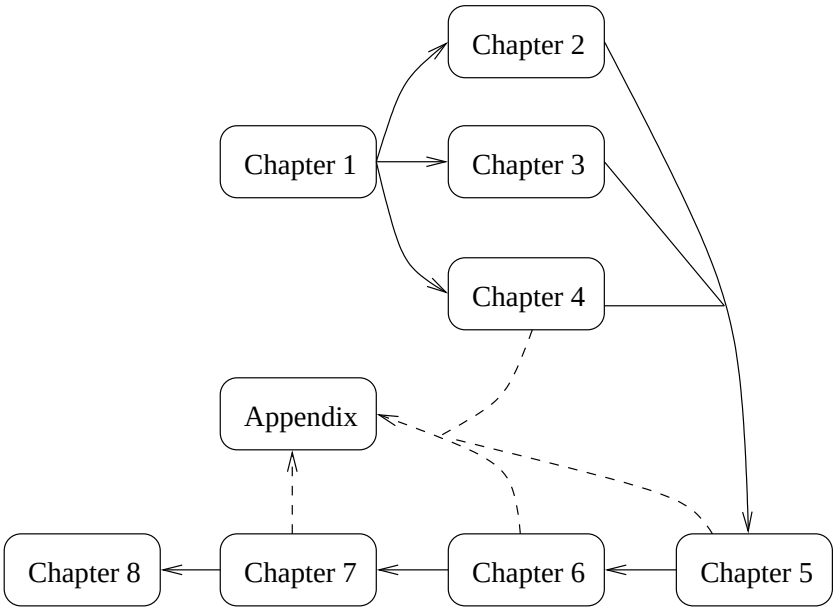
The performance evaluation chapter, Chapter 7, shows how the experiments are run. The chapter shows the results from the performance evaluation as well. For each of the tasks in the evaluation, the results are discussed and explained. At the end of the chapter we sum up the performance evaluation.

The final chapter, Chapter 8, draws a conclusion and sums up the results from the thesis. It also gives a critical assessment of the work we have performed, and looks at the future work; the ideas we have come up with during the project, and would be interesting to investigate further.

At the end of the thesis, the appendices show some of the discussions, tests, queries, plots, and files that were not included in the chapters. Appendix A gives an overview of the acronyms used throughout the thesis. Following, Appendix B shows the tests that are performed to verify some of our points. The next appendix, Appendix C includes some additional information about TelegraphCQ. The succeeding appendices show some of the queries, tables, and plots that are referred to throughout the thesis. Appendix J illustrates the packet headers that are mentioned and discussed in the thesis. Finally, Appendix K shows how the content on the DVD-ROM is organized. Throughout the thesis, we refer to the DVD-ROM, which contains the programs, scripts, and results that were used in the performance evaluation. Appendix K also gives an introduction to practical usage of the experiment setup to perform new experiments, test new tasks, and/or re-test our experiments.

Figure 1.1 gives an overview of the structure of the thesis, and is based in the outline described above.

Figure 1.1 An overview of the thesis, and how it is structured. The dotted lines points the appendices, which are referred to in several chapters.



Chapter 2

Streaming Applications

Traditional databases have been utilized in applications that require persistent data storage and complex querying. Usually, a database consists of a set of records, with insertions, updates, and deletions occurring less frequently than queries. The database system executes the query when it is posed and the answer reflects the current state of the database. However, during the recent years we have seen an emergence of applications that do not fit the data model and querying paradigm of traditional databases [GÖ03]. In these applications, data is better modeled as transient *data streams* than as persistent relations. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, Web applications, manufacturing, and sensor networks. Individual data items in a data stream may be relational tuples e.g. network measurements, call records, Web page visits, and sensor readings [BBD+02]. In the *data stream model*, some or all of the input data that are to be operated on, are not available for random access from disk or memory, but rather arrive as one or more continuous and possibly infinite data streams. Data streams differ from the conventionally stored relation in several ways [BBD+02, PGB+04]:

- The data elements in the stream arrive online and remain only for a limited time in memory. Consequently, the data elements must be handled before the buffer is overwritten by new incoming data elements.
- The system has no control over the order in which data elements arrive in order to be processed, either within a data stream or across data streams.
- Data streams are potentially unbounded in size and may be regarded as open-ended relations.
- Once an element from a data stream has been processed it cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data stream.

- A data stream is append-only, which means it only consists of insertions, and not any deletions or updates.

To integrate data collection and processing, and to enable online (as well as off-line) processing, several research communities have proposed the use of DSMSs for deploying these new streaming applications. Instead of processing queries over a persistent set of data that is stored in advance on disk, a DSMS processes *continuous queries* over the arriving data elements. Continuous queries are evaluated continuously as the data streams arrive. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrive, or they may be produced as data streams themselves. In Chapter 3, we describe the DSMS technology extensively. In Section 3.4.1, we give a more thorough discussion of continuous queries.

Streaming applications may be divided into two different categories: *pull-based* and *push-based* applications. In pull-based applications, data is *pulled* from the data sources into the system when needed, as in traditional database systems. In push-based applications, data elements are *pushed* from the data source into the system. In this thesis, we consider network monitoring, which is a push-based application domain. Consequently, we emphasize the discussion of the push-based domains, with network monitoring in particular. The main pull-based streaming application domain is sensor network, which is the only pull-based domain discussed in this chapter. Pull-based and push-based application domains generate a set of requirements that a streaming application system e.g. a DSMS, should accommodate. At the end of this section, we perform an analysis of such requirements.

2.1 Pull-Based Applications

As mentioned above, the only pull-based application discussed in this thesis is sensor networks. Thus, the following discussion focuses on this application.

Traditional sensors deployed throughout buildings, labs, and equipment, are passive devices that simply transmit signals based on some environmental parameter. Such nodes are for example connected to a local area network (LAN) and attached to permanent power sources. However, recent advances in computing technology have led to the production of a new class of devices: the wireless, battery-powered, computing sensors. These new devices are active computers, capable of not only sampling real-world phenomena, but also filtering, sharing, and combining sensor readings with each other and nearby Internet-equipped end-points. Such sensors may be adjusted in order to allow a suitable degree of precision, for example reporting every second or every fifth second. The sensor nodes communicate via wireless multi-hop radio powered by small batteries [GM04, MF02, YG03] and are made of four basic components: a sensing unit, which is usually composed of sensors and analogue to digital converters (ADCs), power units, a transceiver

unit, and a processing unit. When describing sensor networks, we only consider networks consisting of the wireless sensor type. In Figure 2.1, we illustrate an example of a sensor network. We see that the sensors, which are shown as boxes with an *S* inside, communicate with each other and/or a central node or access point, which is labeled *AP* in the figure. The dotted arrows show communication links. The sensors *pull* data (e.g. light or noise) from the environment based on the functionality of their sensing device, and send the data through the network back to a central node for querying and data analysis. The transmission of data from sensor to sensor towards the central node generates a data stream consisting of sensor readings with the elements arriving at a constant rate. However, this data transmission is very expensive for sensor networks since communication using the wireless medium consumes a considerable amount of energy [YG03]. Since sensors have the ability to perform local computation, part of the computation may be moved from the central node and pushed into the sensor network. Then sensors can aggregate records, or eliminate irrelevant records. Compared to traditional centralized data extraction and analysis, in-network processing can reduce energy and bandwidth consumption by replacing more expensive communication operations with relatively cheaper computation operations, extending the lifetime of the sensor network significantly [YG03]. Based on this structure, sensor networks may be applied on a wide range of applications.

2.1.1 Sensor Network Applications

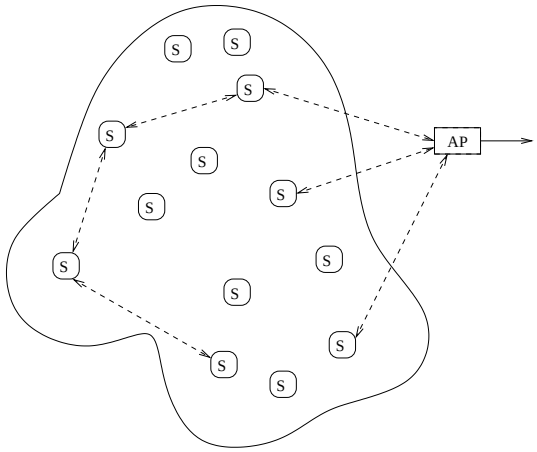
It is predicted that we in the future will see a more extended use of sensor networks, because sensors become smaller and more inexpensive [MF02]. However, already today there are many sensor network applications. Among these are military applications, which through military funding gave birth to many research projects within the field of sensor networks in the early 1980s [CK03]. In the following, we give examples of other sensor network applications.

In a national park, sensor networks can cover large areas over long periods. They can capture micro climates, report unusual seasonal events, and monitor animal behavior. For instance, as part of a project at UC Berkeley [GM04], it was used small sensors to investigate the micro climate at the redwood trees in the UC Berkeley botanical garden. This network played an important role in assisting the botanists in their research and data collection.

Sensor networks can also monitor roads for accidents and traffic hotspots, and warn approaching drivers about the incidents. In such cases, sensor networks can help in diverting traffic, thus increasing transport capacity. Other applications may be to manage road tolling, parking spaces and to detect illegal driving [MF02]. For example, sensors may be located in streets where there is heavy traffic. This may help investigating the driving pattern by registering the number of cars passing by.

As a final example, sensor networks can assist in identifying early signs of fire in

Figure 2.1 An overview of sensors in a sensor network.



forests by helping fire fighters to predict the direction in which the fire is likely to expand. Sensor networks may also assist in rescue operations by locating victims or members of the rescue team.

2.1.2 Limitations of Sensors

Though many new applications have risen following the development of wireless sensors, these sensors also introduce limitations, which constrain their applicability. We list some of the main limitations here:

- *Power* is the defining limit of sensor nodes: it is always possible to use a faster processor or a more powerful radio, but these consume more electricity, which is often not available. Thus, energy conservation is an essential system design consideration of any sensor network application. An example of a sensor is the Berkeley MICA mote [YG03]. The mote is powered by two AA batteries, which provide about 2000 mAh, powering the mote for approximately one year in the idle state and for one week under full load.

- *Communication*: The wireless network connecting the sensor nodes has limited bandwidth [MF02, YG03], latency with high variance, high packet drop rate, and usually provides only a very limited quality of service [YG03].

- *Computation*: Limited computing power restricts algorithmic complexity available to a sensor. In addition, scarce memory resources restrict the amount of intermediate results that a sensor can store [YG03]. Recently, small operating system e.g. TinyOS [CHB+01], and small database systems e.g. TinyDB [MFHH05], have been developed in order to handle these computational limitations.

- *Routing*: For wireless networks, some of the nodes may be mobile in the sense that they are attached to moving objects. In such cases, one has to use special routing algorithms to identify the location of the sensor, maintain a network topology, and verify that the sensor is working as planned. An example of such a routing algorithm is the optimized link state routing protocol (OLSR) [CJ03], which is developed for mobile ad hoc networks (MANETs) [CM99].

2.2 Push-Based Applications

In push-based applications, the system cannot control the rate at which data elements arrive. These applications are concerned with data stream that are often characterized by bursts and heavy load. We discuss three push-based application domains in this section: network monitoring, transaction logs, and financial tickers. The discussion of network monitoring is largely emphasized.

2.2.1 Network Monitoring

In 2002, the Internet consisted of nearly 12,000 Autonomous Systems (ASes). Each AS is a collection of routers and links managed by a single institution, such as a company, university, or Internet Service Provider (ISP) [GR02]. The evolution of the Internet is closely tied to detailed understanding of its traffic. Moreover, tools to analyze Internet traffic are becoming more and more important as the Internet continues to grow rapidly in size as well as in complexity. Hence, operators of large networks and providers of network services need to monitor their network by measuring and analyzing the network traffic flowing through their systems. Network monitoring can provide a valuable insight into the dynamics of network traffic protocols, traffic engineering and capacity planning, congestion and fault diagnosis, and security analysis [HBP+05]. Many monitoring applications are complex (e.g. reconstruct TCP/IP sessions), operate over huge volumes of data (e.g. Gbits and higher speed links), and have real-time reporting requirements e.g. to raise performance or intrusion alerts [CGJ+02]. In a network, one may collect data at several locations (e.g. hosts and routers) both inside the network as well as at the edges. Such data includes [BSW01]:

- Data from network packets and flow traces. Such data may contain information like header fields and packet data. Network packets can be captured by passively listening to the traffic on the network.

- Data obtained by measuring packet delay, loss, and throughput. Such data can be obtained by measuring the behavior of packets that actively are sent through the network.

- Router forwarding tables and configuration data. The routers in the network send data packets to each other describing network characteristics and routing information. Such data can be used to get a total overview of the traffic at several routers. An example is the Simple Network Management Protocol (SNMP) [CFSD90]. Data from this protocol is used to communicate network information between the gateways and the network administrators. Broadly, network traffic analysis can be divided into three tasks:

1. Collecting the data e.g. router configuration data.
2. Measuring the collected data e.g. to obtain statistics from the collected data.
3. Analyzing the measured data e.g. to characterize and model traffic in various layers.

In this section, we focus on task two and three. Firstly, we consider network traffic measurement, whereas secondly, network traffic analysis is discussed.

Network Traffic Measurement

Network traffic measurements play a crucial role in providing operators with a detailed view of the state of their networks. These measurements are conducted on a continuous basis and the results are compiled into reports for management that are used in management decisions on various time scales. Traffic measurements are divided into two different techniques; *passive* and *active* measurements.

When using the passive technique, one simply observes and records the traffic as it passes by. This approach measures real traffic, is useful for characterizing the Internet traffic, and does not disturb the network traffic by adding extra load. However, one does not have full control over the measurement process [Sie06]. When using the active technique, one injects packets into the network, monitors them, and measures the services obtained. This technique is useful for inferring the network characteristics, and one obtains complete control over the measured traffic. However, this technique may disturb network traffic by adding extra load [Sie06].

In addition to the active and passive techniques, we may divide network measurement into two different approaches; online and offline measurements. With online measurements the traffic is captured and the data is measured in a real-time manner, whereas with offline measurements traffic is captured into trace files for later measurements.

Examples of traffic measurement tasks, adapted from the STREAM query repository [Bab02], are:

- For each source IP address and each five-minute interval, count the number of bytes and number of packets resulting from HTTP requests.

- Find the source-destination pairs in the top five percentile in terms of total traffic in the past 20 minutes over a backbone link B.
- Generate the flows in the packet stream, and for each flow, output the source and destination addresses, the number of packets constituting the flow, and the length of the flow.

Other examples are:

- For each source IP address, count the number of active flows from that address in each five-minute interval. A flow may be defined as all packets that have the same source and destination IP addresses, where successive packets have an inter-arrival time less than 30 seconds.
- Maintain the fraction of packets on a particular backbone link B generated by a particular customer network C in the past hour.

Network Traffic Analysis

In network traffic analysis, we use the measurements to maintain network state, to detect causes of problems in the networks, and in capacity planning and optimization. Broadly, network traffic analysis can be divided into three different categories: *Traffic characterization and modeling*, *network characterization and modeling*, and *anomaly detection* [Sie06].

With respect to traffic characterization and modeling, network traffic packets may be recognized based on the characteristics of the protocols they use at different layers. At the application layer the traffic may be related to e.g. peer-to-peer file-sharing applications and Skype. At the transport layer traffic is typical related to TCP and UDP.

After the widespread usage of peer-to-peer (P2P) networking during the late 1990s, P2P applications have multiplied. Their diffusion and adoption are witnessed by the fact that P2P traffic accounts for a significant fraction of Internet traffic [SLP05]. Furthermore, there are concerns regarding the use of these applications, particularly when they are employed to share copyright protected material. In addition, many ISPs are reluctant to let customers consume bandwidth in the file sharing operations, which is a part of P2P applications. It is important to gain a deeper understanding of the characteristics of P2P traffic, because it accounts for a significant part of the Internet traffic. Such knowledge may be valuable in further development of P2P applications or new protocols. However, P2P traffic must be identified before it may be measured and analyzed. Identifying P2P traffic may not be easy, because it for instance may camouflage by using TCP ports that are not usually utilized for P2P traffic.

Another protocol that it is important to understand in a best possible manner is TCP, because it carries over 90 % of the network load in the Internet [Sie06].

As for P2P applications, TCP traffic must be identified before it can be measured and analyzed. Identifying TCP packets is straightforward, because the IP header provides this information in one of its fields[1]. However, as indicated in Chapter 5, recognizing TCP connections is not a trivial task, particularly not online.

Managing a large network is a complex task, which may be conducted by a group of human operators. With respect to network characterization and modeling, these operators track the characteristics of the network to detect equipment failure and shifts in traffic load. Network characteristics and modeling may be based on parameters such as e.g. topology, utilization, packet delay, and packet loss. By joining SNMP data and/or configuration data from different network elements, it is possible to obtain network topology, and by aggregating packet traces or SNMP data, it is possible to maintain statistics of link and router utilization. Packet loss, per-hop and end-to-end delays, and network throughput are measured by either joining packet traces collected from multiple points in the network, or using a dedicated system that generates network traffic to measure these parameters [BSW01].

Traffic engineering is concerned with performance optimization of traffic-handling in operational networks. When optimizing performance, it is important to minimize over-utilization of capacity when other capacities are available within the network. Updated information on network characteristics is important in order to detect problems with network traffic. After detecting and troubleshooting a problem, operators may change the configuration of the equipment to improve utilization of the network resources and the performance experienced by end users. An example of a performance problem is link congestion. A link may be congested because of an increase in demand between some set of source-destination pairs or a failed link or router in a network causing changes in routes. One way to detect link congestion is to calculate utilization statistics from SNMP.

For an ISP it is important to have knowledge regarding characteristics of bandwidth consumption in order to make proper allocation of resources or to decide where and when to install new equipment. Examples of decisions to make are where to put the next backbone router, when to upgrade a peering link to higher capacity, and whether to install a caching proxy for cable modems.

The final category is anomaly detection. The widespread usage of the information-sharing possibilities provided by the Internet has revolutionized our society by enabling us to communicate easily with people around the world, and to access and provide a large variety of information-based services. However, this success has also enabled the use of the Internet in ways that are considered hostile. *Spam*, *viruses*, *worms*, and *Denial-of-Service* (DoS) attacks are well known terms today. As the number of network-based attacks increase, and the variety and sophistication in these attacks grow, early detection of potential attacks will become crucial in reducing the impact of these attacks. We show some examples of anomalous activ-

[1] An overview of the IP, TCP, as well as Ethernet and UDP packet headers is located in Appendix J.

ity by describing denial of service (DoS), worms and viruses, and the probing for vulnerability.

DoS is characterized by an explicit attempt by attackers to prevent clients from using a service. DoS has been among the most common form of Internet attacks. The basic form of a DoS is to consume scarce computer and network resources, such as kernel data structures, CPU time, memory and disk space, and network bandwidth [JMSS05]. An example of a DoS is the TCP SYN flood attack, which exploits the 3-way handshake used to establish a TCP connection [Pos81b, JMSS05]. In a normal scenario, a sender initiates a TCP connection by sending a SYN packet, i.e., a packet having the SYN bit set. The receiver responds with a SYN/ACK packet, and the sender completes the 3-way handshake by sending an ACK packet. Following the sending of the SYN/ACK packet, the receiver allocates connection resources (kernel and data structure) to remember the pending connection for a pre-specified amount of time. The attack occurs when the attacker repeatedly sends SYN packets, typically with different source addresses, causing the receiver to deplete its connection resources, preventing service to other users. In principle, the attack can be identified by measuring and analyzing the number of SYN packets for which a SYN/ACK packet is sent, but no correlating ACK packet is seen within a given delay.

A *worm* is a self-propagating malicious code, which exploits vulnerabilities in the underlying operating system to inflict its damage, and to replicate and propagate itself [JMSS05]. A virus, on the other hand, relies on user actions for its propagation, and hence tends to spread slowly. Payload and specific mechanism of propagation may identify known worms. For example, activity of the Slammer worm is identifiable in a network by the presence of 376 bytes UDP packets, destined for port 1434/UDP of SQL Server [JMSS05].

We see that attacks exploit known vulnerabilities in services. A typical precursor to attacks is the identification of machines that have specific services available, and hence can potentially be exploited. This takes the form of an attacker probing for open ports on a set of host machines. To determine if a port is open, an attacker sends a packet to a host attempting to connect to the specific port. If the target host is listening on that port, it will respond by opening a connection with the attacker. This implies that during the probing phase, the attacker would not spoof the IP source address [JMSS05], meaning that such anomalous activity can be detected by measuring and analyzing the number of distinct <destination IP, destination port> pairs with the same source IP address.

2.2.2 Transaction Logs

Massive transaction streams introduce a number of opportunities for data mining techniques. Examples of transactions are calls on a telephone network, commercial credit card purchases, stock market trades, and HTTP requests to a Web server

[CFPR00]. The goal is to find interesting customer behavior patterns, identify suspicious spending behavior that could indicate fraud, and forecast future data values [GÖ03]. A transactional data stream is a sequence of records that logs interactions between entities. For example, a stream of credit card transactions contains records of purchases by consumers from merchants. Data mining techniques are needed to exploit such transactional data streams since these streams contain a huge volume of simple records, any one of which is rather uninformative unless it is part of a total overview [CFPR00]. However, when the records related to a single entity are aggregated over time, the aggregate can yield a detailed picture of evolving behavior, in effect capturing the "signature" of that entity. A signature for a phone number might contain directly measurable features such as when most telephone calls are placed from that number, to what regions the calls are placed, and when the last call was placed. Queries investigating these matters may be quite similar to those detecting anomalous activity in the Internet. It might also contain derived information such as the degree to which the calling pattern from the number is "business-like" [CFPR00]. Such information is useful for target marketing and for developing new service offerings. Other examples of transaction log analyzing tasks, which are adapted from [GÖ03], are:

- Find all pages on a particular Web server that have been accessed in the last fifteen minutes with a rate that is at least 40 % greater than the running daily average.
- Examine server logs and re-route users to backup servers if the primary servers are overloaded.

Other examples are:

- Track mobile phone records and for each mobile phone number, determine the number of
  1. Distinct base stations used during one telephone call.
  2. Bytes transferred in order to open Web pages using the wireless application protocol (WAP) [MDK+00].
  3. Bytes transferred in order to download e.g. ring tones, games, and wallpapers.

2.2.3 Financial Tickers

In the United States, up to 100,000 quotes and trades (ticks) are generated every second [ZS02]. This results in a stream of stock market transactions, which consist of buy or sell orders for particular companies from individual investors. Online analysis of streams of financial tickers might help a stock market trader to discover correlations, identify trends and arbitrage opportunities, and forecast future values.

Traderbot, a typical Web-based financial ticker, allows its users to pose queries such as the following [GÖ03]:

- High Volatility with Recent Volume Surge: Find all stocks priced between $20 and $200, where the range between the high tick and the low tick over the past 30 minutes is greater than three percent of the last price, and where in the last five minutes the average volume has surged by more than 300 %.
- NASDAQ Large Cap Gainers: Find all NASDAQ stocks trading above their 200-day moving average with a market cap greater than $5 Billion that have gained in price today between two and ten percent since the opening, and are within two percent of today's high.
- Trading Near 52-week High on Higher Volume: Find all stocks whose prices are within two percent of their respective 52-week highs that trade at least one million shares per day.

2.3 Requirements Analysis

In this section, we analyze the requirements that are imposed by the application domains mentioned in the previous sections. Firstly, we consider the common requirements of these applications. Secondly, we discuss the requirements raised by sensor networks and network monitoring. Recall that there is a major difference between standard database sources and the data sources for the network monitoring applications. Network monitoring applications have to handle data streams, i.e., data elements, or tuples, that are continuously produced and pushed into the system. One important requirement raised by streaming applications is that queries over such data streams need to be processed immediately, in real-time. This because it is expensive to save such large amounts of data to disk and much of the data may not be of interest later. Moreover, the streams represent real world events that need to be responded to. Generally, all streaming applications need a system to handle large amounts of arriving data packets. If not all the data is considered interesting, it needs functionality for choosing only the packets that are most important with respect to the application. In cases of much important data tuples, the system some times has to aggregate on the streams in such a way that only the most representative tuples or averages of the data results are displayed. In addition, the systems have to respond quickly to sudden changes in the data streams and register or output these changes. In handling this, a set of general requirements for streaming applications emerge. In the following, we list these requirements, many of which are collected from [GÖ03].

- *Continuous queries*: To analyze a large range of behaviors attached to the different applications, one would need to collect data on an ongoing basis

rather than as a one-time event. Hence, it is required that a system is capable of processing data in a continuous manner. This means that the query has to be started and stopped explicitly by a user or by a system. If not stopped, it is assumed to run infinitely.

- *Projection*: To reduce the size of queues in memory and in turn improve memory utilization, it is required that a DSMS supports projection, i.e., choosing only a subset of the attributes in a relational tuple.

- *Selection*: All DSMSs require support for complex filtering. The selection should manage to fetch only data having certain values, such that much data can be excluded from further processing at an early stage. As an optimizing factor, it should be possible to push both selections and projections as close to the source data stream as possible.

- *Joins*: In order to perform a wide range of analyzing tasks, a DSMS should include support for joins between multiple streams and joins between streams and stored relations. By supporting this requirement, the DSMS may analyze the data to find patterns that depend on correlations between many streams and relations.

- *Aggregation*: By supporting aggregations, the DSMS may attach statistics to application dependent patterns that it recognizes within streams and/or relations. The aggregations, which calculate sums, maximums, minimums, counts, and averages, may also assist in obtaining an overview of the data values in the stream.

- *Windowed queries*: Many operators (e.g. aggregating operators) are blocking, i.e., the operator must see all input data before it can produce any output. However, data streams are considered infinite. Therefore, the DSMSs have to support some type of partitioning over the streams, such that blocking operators may process data within such partitions or windows. If windowing is not supported, blocking queries can never be performed correctly, since the DSMS needs to see all tuples of the stream in order to compare them.

- *Processing multiple queries*: In many scenarios, multiple users pose similar queries over the same data streams. Since streams are append-only, there is no reason that a particular data item should not be shared across many queries [MF02]. Hence, a DSMS should support multiple, concurrent queries.

- *Sub-queries*: To analyze characteristics of an application, the DSMS should be able to perform complex queries to identify mechanisms within the application. In order to perform such complex queries (e.g. reconstructing TCP connections) support for sub-queries is required. Sub-queries may appear in several different query clauses, for example selections and more complex projections.

- *Nested aggregation*: Complex aggregates, including nested aggregates (e.g. comparing a minimum with a running average) may be needed to compute trends in the data sets. A nested aggregate is an expression

$$agg_n(agg_{n-1}(...(agg_0(X))...)),$$

  where each agg_i is an aggregate function and $n \geq 1$ [JC99]. Nested aggregations must be calculated continuously within windows.

- *Multiplexing and demultiplexing*: This requirement can be viewed as the group-by aggregation and union set operator, respectively. The multiplexing and demultiplexing can be used to decompose and merge logical streams, depending on the answers required.

- *Frequent item queries*: These are known as *top-k* or *threshold queries*. This means that the DSMS only query for items that appear frequently, and may be of greater importance than other items. This may, in addition, be part of the selection such that for instance only the tenth tuple, or values over a certain threshold are selected.

- *Stream mining*: Operations such as pattern matching, similarity searching, and forecasting are required for on-line mining of streaming data. The mining thus relates to a more experience-based and intelligent way of running the queries. If, for example, a query in a weather monitoring network is told to report data only when it is rainy, it might have to compare its input data to historical data and other observations to get assistance in the decision process.

- *Adaptive query processing*: A fundamental challenge in many streaming applications is that conditions, for example data values, may vary significantly over time. Since queries in these systems are usually long running, or continuous, it is important to consider adaptive approaches to query processing. Without adaptivity, performance may drop drastically as stream data and arrival characteristics, query loads, and system conditions change over time [BW04].

The above list is used throughout the thesis as a reference. We show how the DSMSs and TelegraphCQ implement these requirements. Following, is a short additional list of requirements for the pull- and push-based data stream models, exemplified with sensor networks and network monitoring, respectively.

2.3.1 Sensor Networks

The main limitation in sensor networks is based on power consumption, which provides some additional requirements to the application. As sensors are part of

a pull-based stream model, the queries are required to specify the pull interval, i.e., identifying when to sample data from the environment. This interval has to be relative to the power consumption. It is also required that sensor networks distribute queries among the nodes to reduce the amount of data, because sending data through wireless links consumes much power. This means that each node plays a part in the total query processing. For example, the node may perform some simple aggregations on the data before sending it to another node.

In some sensor network monitoring applications, it may be necessary with a large-scale deployment of sensor nodes. A large number of nodes may even require more scalability in cases where additional nodes may be inserted into or removed from the network.

2.3.2 Network Monitoring

It is required that the DSMS provides good approximation techniques in order to keep the query answer as correct as possible, because the data streams may arrive at high network loads. An example of an approximation technique is load shedding, i.e., dropping elements from query plans and saving the CPU time that would be required to process them to completion [ABB+04].

The load on a network usually consists of traffic belonging to many network protocols. Thus, another requirement imposed by network monitoring is that packets belonging to different protocols should be processed by the DSMS. Additionally, it is important that the DSMS supports the different data operators that may be required in a packet header. An example is the IPv4 address, that consists of four numbers separated by dots. Moreover, a data packet may offer complexity with regard to a varying number of header fields. For example, both the IPv4 [Pos81a] (henceforth IP) and the TCP [Pos81b] headers have option fields, which contain optional information, such as information about the TCP's maximum segment size (MSS). A DSMS should provide functionality for supporting these variations. Another example is the extension headers in IPv6, which amongst others contain routing information.

Some of the network protocols tend to be complex. For instance the TCP standard specifies how two nodes should act when they establish and close down a connection [Pos81b]. Thus, when measuring and analyzing network and protocol behavior, it is required that the DSMS manages to reflect protocol states in both the network and the network nodes. Hence, the declarative language provided by a DSMS should provide a wide range of operators in order to express queries that may be used when monitoring protocol behavior.

Chapter 3

Data Stream Management Systems

3.1 Introduction

This chapter describes and discusses some of the main issues in data stream management systems (DSMSs). A DSMS is a system that poses queries on a stream of data. Based on the requirements in Section 2.3, we show how these are designed in the DSMSs. Throughout the thesis, we use the following definition of a stream, as taken from [AW04]. Given the discrete ordered time domain $\mathcal{T}$:

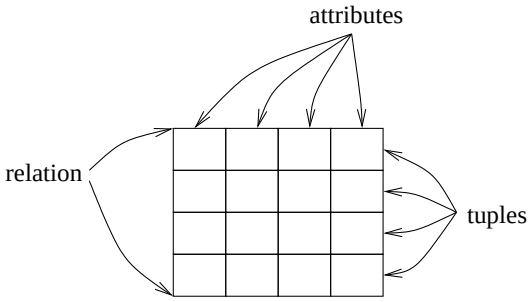
Definition 1 *A stream S is a possibly infinite bag (multi set) of elements $< s, \tau >$, where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the time stamp of the element. There is a finite but unbounded number of stream elements for any given time stamp $\tau \in \mathcal{T}$.*

As mentioned in Chapter 2, several applications motivate for a system that is able to easily and in real-time extract relevant information from data streams. Examples of such applications are *sensor networks*, *financial tickers*, *transaction log analysis*, and *network traffic monitoring*.

As stated in Chapter 1, we have chosen to focus on network traffic monitoring. Therefore, the description of the DSMS focuses on this application.

DSMSs are often compared to database management systems (DBMS) [GMUW02] since both deal with querying of data. The following section gives a short review of the DBMS's main issues, terms, and characteristics. The differences are shown by comparing the two systems. We further discuss several of the DSMS requirements listed in Section 2.3.

Figure 3.1 An overview of the terms and the tuples in relations.



3.2 Database Management Systems

One of the DBMSs' most important tasks is to efficiently and quickly obtain data from a storage device, e.g. a hard disk [GMUW02]. Examples of such data can be employees in a company, or packet traces from a network [SBG+05].

The data is mostly represented using *relations*, or tables[1]. Based on [GMUW02], we describe a relation as a two-dimensional array for representing data.

In the following, we explain some of the terms that are used to describe certain elements in the DBMS's relations. These terms have already been mentioned several times in the preceding chapters since they describe the equivalent elements in the DSMS, e.g., sequences of data items, as well.

The relational representation that we choose to focus on models the data in columns and rows, i.e., *tuples*. Each of the items in a tuple is called an *attribute*. The tuple has a *fixed* and predefined amount of attributes. To get an overview, Figure 3.1 illustrates the terms. The relations are stored in *blocks* on the storage device. Relations can be *joined* using e.g. similar attribute values. By join, we mean *natural join* or *theta join*. Such joins are produced by performing a Cartesian product and filtered by a set of given conditions [GMUW02]. There are several main characteristics that describe a DBMS [BBD+02, GP]:

1. *Persistent storage*. When a user stores data on the disk, he or she wants the data to stay there, unless explicitly deleted. If the data is deleted, it is assumed that this is done on purpose. Having a persistent storage, the user wants to manipulate, delete and observe the data in the database. The user may also want to verify that nothing has been changed unintentionally.

2. *One-time transient queries*. When the user wants to collect data from the database, a *query* is run once and the system outputs the results. Hence, the DBMS has a *programming interface* that the user interacts with. The query

[1] "Table" and "relation" are used interchangeably throughout the text.

is rewritten, i.e., parsed, transformed, and optimized by the system to obtain the correct data.

Figure 3.2 shows the main rewrites. First, the user has an idea of what he or she wants. Then, this is written to the computer as a query. Usually, this query is expressed using a *query language*, and, as mentioned in Chapter 1, SQL (Structured Query Language) is most commonly used [GMUW02]. A general structure of an SQL statement is given as follows:

```
SELECT <attributes>
FROM <realtions>
[WHERE <conditions>
  [GROUP BY <attributes>
    [HAVING <quality>]]
]
```

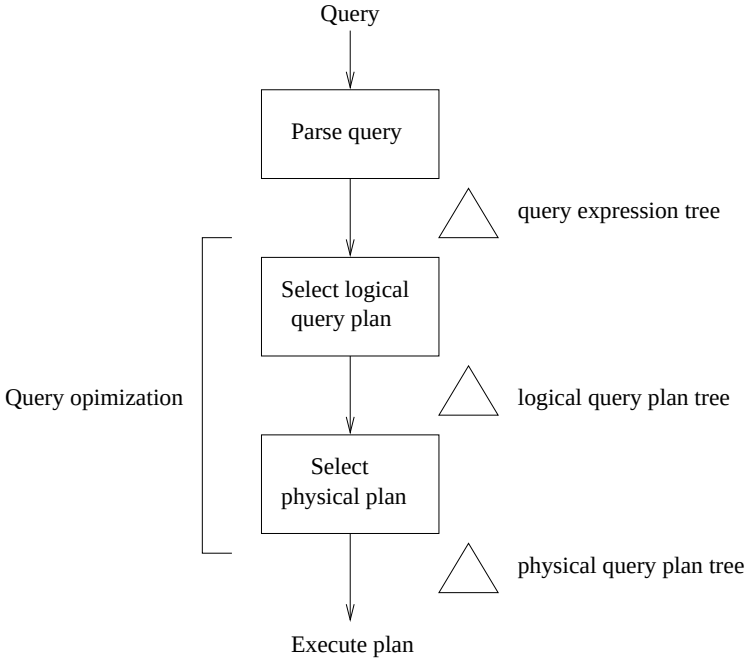
If the query is only filtering the data stream without joining, it is sufficient to use the `SELECT`-, `FROM`-, `WHERE`-clauses. As shown in the general structure, the `SELECT`-clause provides the projection of attributes, the `FROM`-clause tells which relations to obtain tuples from, and the `WHERE`-clause selects the tuples that have certain values.

SQL gives the opportunity to select data from the database in a desired format, i.e., attribute number, types, and order. The query is then parsed and transformed into an expression tree. At the next stage, a logical query plan is constructed, e.g., which operators to use, are chosen. The operators are then put into a logical query plan tree. This tree may be optimized by the system after certain rules, for example by analyzing the size of the relations and number of attributes queried for, or by using heuristics about relations if there are no meta-data about the relations available. The system finally turns the logical query tree into a physical plan, i.e., choosing the algorithms for the operators.

3. *Random access*. When querying certain relations, the DBMS manages to access data as long as it is available on the storage device. This also means that the data can be accessed by specifying which block numbers the relations are stored in. If the tuples are indexed, even these can be accessed directly. Since the data is stored persistently, knowing where the data is located is sufficient for retrieval.

4. By claiming persistent storage, it is expected that the storage space has to be viewed as *unbounded* or *infinite* by the DBMS. This is because the system should be able to store new tuples without needing to delete older data. The idea is also that the system should not have any pre-fabricated limit of storage, even though this is the case when using a device with limited storage capabilities. If a disk is full, a new disk is simply added.

Figure 3.2 An example of query rewrites [GMUW02].



5. DBMSs often provide a multi-user environment. This environment has to make sure that the database data is written by only one user at the time, such that inconsistent states can not be found. In case of system errors, the DBMSs may also need to re-create the last working state. This is managed by logging all critical operations, and is handled by the *transaction manager*.

6. *Only current state matters*. The DBMS does not use any historical data for e.g. optimizing the queries. This is not equivalent with the previous point where the transaction manager provided an overview of the transactions over time. As already mentioned, a query is optimized after certain rules and heuristics, based on knowledge about the size of the relations used in the query, for example. The main optimizing factor is the number of disk I/O. Relative to accessing the main memory, disk I/O is time consuming, and thus expensive, operations that have to be kept to a minimum.

7. The traditional DBMS does not support any *real-time services* such as having deadlines determining *when* the data should be output. This denotes that the system does not throw away tuples if they exceed a certain time limit due to a complex query or large amounts of data. If a query is complex, the time it takes to get a result increases with the complexity.

8. It is assumed that a DBMS returns the *precise* answers when queried, because of the prior capability. This may, as stated in the prior item, affect the

time consumption and hence reduce real-time support.

9. Since the DBMS's main focus is to return the correct data from the database, the update rate, i.e., the *rate for input of data is relatively low*. This is because the system focuses on storing the data in such a manner that the retrieval is as efficient as possible. An example is to store a relation on continuous blocks on the hard disk. This may force to moving existing blocks, but may ease computation time if the query has to e.g. run through the whole relation in a query. Another example is to spend time in indexing the relations and thus changing old indexes.

As shown above, there are several points that describe the DBMS. In the following, we discuss one part of the DBMS that helps fulfilling the requirement of precise results. These techniques have been developed to help the queries process correctly for relations that are too large to fit into main memory, and thus need a considerable amount of disk I/O. These techniques are called *one-pass*, *two-pass*, and *multi-pass* algorithms.

A query *operator* is for example an equi-join between two relations R and S on an attribute a, which we denote $R \bowtie_a S$. We use the equi-joins as an example even though there are several other operators that may require n-pass algorithms. Understanding the *n-pass* algorithms helps understanding the complexity that may occur for large amounts of data.

One-Pass Algorithms

One-pass algorithms are used when one of the relations fits into the memory. The available capacity of the memory is M. The relation R is size $B(R)$ blocks, and

$$B(R) \leq M - 1.$$

The means that $B(R)$ is equal or smaller than the available memory minus one memory block. If the DBMS uses an operation that joins the two relations, $R \bowtie S$, the other relation S is written block by block into the remaining memory block. In the memory, the tuples are joined with the correct tuples of R, based on attribute equalities, and sent to output. Figure 3.3 illustrates the one-pass algorithm; the lower block is the one used to filter the relevant tuples from S.

Two-Pass Algorithms

The idea behind the two-pass algorithms is that it handles relations of size

$$M < B(R) \leq M^2.$$

In these cases only parts of R are inserted into memory, *sorted* on some predefined criteria, e.g. an attribute's value, and then written back to the hard disk. This is

Figure 3.3 The one-pass algorithm.

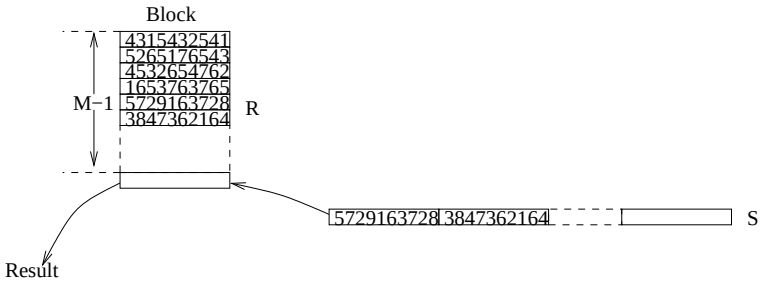
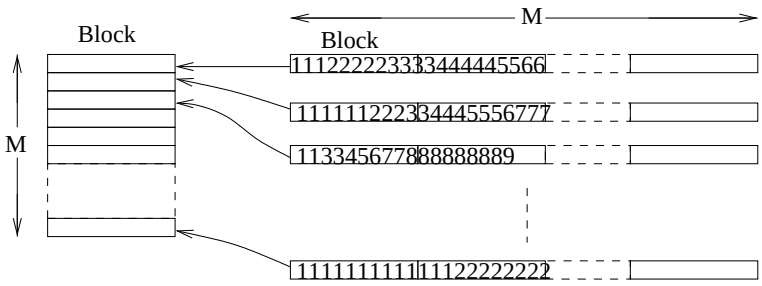


Figure 3.4 The two-pass algorithm



the *first pass*. In the *second pass*, the operation is performed on the relations. The memory is then filled with up to M sorted blocks at the time. These are joined together. Figure 3.4 gives an illustration of how the two-pass join algorithm works. As shown in the figure, when joining, each of the relations are sorted after the joining attributes, and inserted into memory before they are joined on equality, for instance. One solution is to *merge* the relations. A consequence is that if for example S contains many 1s, the other relations wait until a new value shows up.

Multi-Pass Algorithms

The multi-pass algorithms are used when

$$M^2 < B(R).$$

These are more complex, hence forcing considerably more disk I/O. As the two-pass algorithm sorts the relations into $O(M)$ one-pass relations, the multi-pass algorithm recursively sorts into $O(M^2)$ two-pass relations, which again sorts up to one-pass relations.

These three algorithms show how the size of a relation affects the storage device and how much data it is forced to send and receive for correct results.

3.3 Data Stream Management Systems

The characteristics of the DBMSs are shown in Section 3.2. The DSMSs focus on querying *streams of data* instead of the database. As mentioned in the preceding chapters, this causes some changes in the DSMSs' architecture compared to the previously mentioned list [BBD+02, GP]. Following, we compare the two systems' design to show the differences between them.

1. Instead of persistent relations, the DSMS aims to handle *transient streams*. This implies that disk storage is not an issue; the data enters and leaves the system at possibly high rates. Though, as this poses an architectural opposite to the DBMS, some DSMSs have integrated a DBMS as a part of it. This opens for the possibility of joining between streams and tables, for instance. An example is TelegraphCQ [CCD+03].

2. When the DBMSs access the data once for each query, the DSMS uses *continuous queries*, which are queries that continuously obtain tuples from the streams. In other words, the DBMS supports transient queries over persistent data, while the DSMS supports persistent queries over transient data.This issue is described thoroughly in Section 3.4.1.

   Since the queries are continuous over a set of streams, the DSMS also allows for several queries to run concurrently. In a network monitoring scenario, an example would be several concurrent queries that aim to obtain different information from the network.

3. *Sequential access*. Since the data arrives as a stream, the DSMS reviews the tuples as a linear sequence, and does not have access to the data before or after the access interval. The DBMS data is stored in a database, and the data can be randomly accessed by specifying which blocks to read. In operations like aggregations and joins, this means that the DBMS *blocks* while performing these operations. This is not possible over a linear stream of data. When operating over streams, the DSMSs have to support *windowing* for blocking operators, which means that even though the stream is infinite, calculations are performed on small partitions of the stream. These partitions are located in main memory, which implies limitations with regard to window size and accuracy. This is discussed in Section 3.4.1.

4. Since the DSMS does not generally aim to store tuples as they arrive the system, it is bound by the size of the available *memory*. This is also an issue with regard to disk I/O. As the streams may arrive at high rates, expensive and time consuming disk I/O can not be allowed. This means, as deducted in the description of the n-pass algorithms, that only *one-pass* algorithms can be used.

5. Due to optimizing, the DBMS has a set of rules that are introduced in the query rewriting process illustrated in Figure 3.2. An example of an optimizing rule is to push projections as far as possible down to the source, i.e., the database or the data stream. By doing this, less attributes are sent to the next operators, hence reducing the load. These rules also play a role in the DSMS, but the DSMS needs to adapt to the stream as well, by optimizing the query tree on the fly, or re-allocating queue sizes.

6. The DSMS is pledged to compute data and deliver results *within a deadline*, i.e., a time-limit. As data streams may arrive at high transfer rates, this may force the DSMS to throw away tuples that it can not compute, because of factors such as too complex queries. Generally, this means that not all the tuples may be computed, and that the DSMS has to support a set of *approximation algorithms* that deliver results that e.g. give a *sample* of the discarded tuples. DBMSs can not guarantee that the results - since they are required to be correct - will be displayed within a deadline. Complex queries using relations of several Gbytes may take hours to terminate in a DBMS.

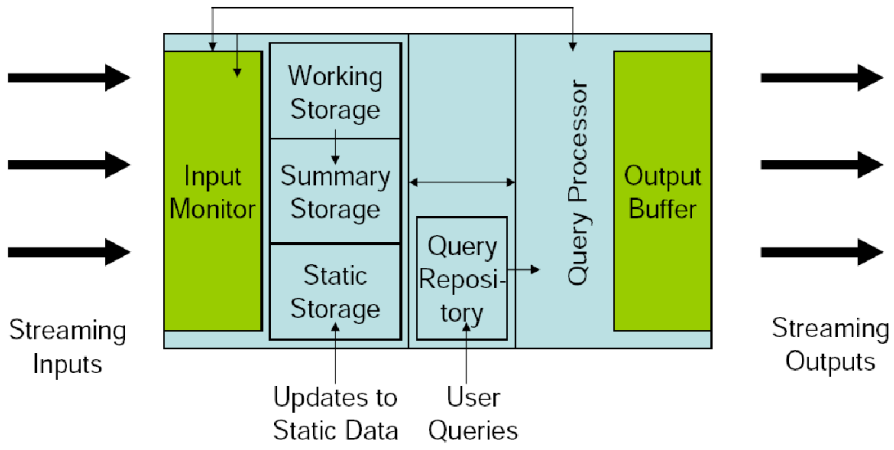
   The update rate also plays an important role in real-time processing. Compared to main memory processing, the disk-based update rate is limited due to the relatively slow hard disk. This means that the DSMS only gets the data from the network, performs computations on them, and then deletes them, to make room for new tuples.

The main building blocks of a DSMS are illustrated in Figure 3.5. One or more streams enter the system and are processed by the query processor. State information might be sent between the input monitor and the query processor to inform about e.g. stream characteristics, such that optimizing and adaption may be performed. When the query operators are finished processing the tuples, the result is sent to the output buffer.

3.4 Issues in Data Stream Management Systems

The preceding sections give an overview and introduction to DSMSs, and also the main differences between DBMSs and DSMSs. Some of the most important issues - as shown in Chapter 2.3 - are described and discussed in the following sections. Firstly, Section 3.4.1 discusses some of the challenges that are associated with continuous queries and windowing. Section 3.4.2 discusses approximation and techniques for optimizing the queries. Section 3.4.3 gives a short overview of query language qualities and usage. Finally, Section 3.5 gives a short description of some of the existing DSMSs to date.

Figure 3.5 A generic overview of the general DSMS architecture [Mou].



3.4.1 Continuous Queries (CQs) and Time Windows

This section starts by showing how CQs have evolved from their first appearance in append-only databases to how they are used in DSMSs today. An append-only *database* is a database that only adds tuples, without ever deleting or updating them, and research on CQs allegedly started on such database systems [GÖ03].

Terry et al. [TGNO92] state that sometimes a user wants to receive the tuples exactly when they are inserted into the database. Such tuples can have time critical values, and are only valid or interesting for a short period of time. Thus, there are requirements for a mechanism that supports such functionality. Based on these requirements, [TGNO92] introduces the term *continuous queries* (*CQs*), i.e., queries that continuously search the database to see if any new entry or tuple has arrived, and possibly report the results if the tuples matches certain criteria.

The CQ's Building Blocks

CQs raise other issues than transient queries. To query the data stream, several suggestions on semantics have been proposed [AW04, KS05, TGNO92]. Throughout the thesis, we use the definition stated in [TGNO92]:

Definition 2 *Continuous semantics: the results of a continuous query is the set of data that would be returned if the query were executed at every instant in time.*

Formally, this means that if $A(Q,\tau)$ is the set that is returned by running the query Q over a data set at time τ, $A(Q,\mathcal{T})$ denotes the union of all sets over a time interval $\tau \rightarrow \mathcal{T}$, then [GÖ03]

$$A(Q,\mathcal{T}) = \bigcup_{\tau=1}^{\mathcal{T}} (A(Q,\tau) \setminus A(Q,\tau-1)) \cup A(Q,0). \tag{3.1}$$

Equation 3.1 shows that only the newest arriving tuples are handled by the query. This equation assumes that the database has a *monotonic* behavior.

Monotonicity is defined by a preservation of order. Thus, for a monotonic function f, iff $x \leq y$, then $f(x) \leq f(y)$. For a database, this may not always be the case. Sometimes tuples are changed after they are inserted. Therefore, a *non-monotonic* query may sometimes give a more correct result [GÖ03]

$$A(Q, \mathcal{T}) = \bigcup_{\tau=0}^{\mathcal{T}} A(Q, \tau). \tag{3.2}$$

The problem with the non-monotonic query is that it has to query the whole data set for each round. Given a CQ, this means that the whole set has to be queried continuously. With an increasing set, i.e., tuples are added to the system without being deleted, the time that is consumed each time the query is run will increase linearly.

The major challenge in using the semantics in Equation 3.1 is to preserve this monotonicity over a continuous stream in a deterministic way. By deterministic, we mean that when the query has started, it will always give the same results over the same data stream. [TGNO92] illustrates this by an example:

> A database contains messages that are inserted each time stamp t. We want to get all the messages that have *not* been replied to.

This example shows that without any more information, this query may force the system to *non-monotonic* computing; When do we know that we have got all the answers? Without any precision, this query returns all the messages as they arrive, since a message can not be replied to before it is sent. If the database is append-only, we eventually get the correct results as more messages arrive, if the query runs forever. The system needs to view all the previous messages to see if the new message is a reply to an older one. The problem is that an incoming data stream, as defined in Definition 1, may require a vast amount of storage. Hence, the database has to drop tuples to make the querying possible. This means that there is a risk of deleting messages before the replies appear.

Such an inconsistency leads to the term of *windowing* over a stream of data. A window is a partition over the stream that guarantees determinism inside it. This means that, given a plain limit, the query stated above can be rephrased to

> Get all the messages that have not been given a reply within five minutes after it has been sent.

The query looks at all the messages and stores them. It also reduces the monotonicity by only re-calculating over a five minutes window, which again returns the

correct tuples. Still, there has to be re-calculations, something that leads to *blocking* of the system over the time window.

Blocking may be critical for the DSMS. A blocking operator is a query operator that is unable to produce the first of its output until is has seen its entire input [BBD+02]. For example, the DBMS blocks while it performs two- or multi-pass operations; the data is obtained from the database, e.g., sorted, and then written back to disk possibly several times before the final result is returned. This is possible when there is a finite bag of data that is handled. As pointed out in the stream definition, a data stream is an *infinite* bag, thus reducing algorithms only to support one-pass.

In the context of joining, and that we still include the database in the DSMS model, it is possible to join between streams, and to join between streams and relations[2]. For values between 1 and n we denote $S_1 \dots S_n$ for n streams and $R_1 \dots R_n$ for n relations. We set i and j such that $1 \leq i \leq n$ and $1 \leq j \leq n$. In case of $S_i \bowtie S_j$, it is required that $B(S_i) + B(S_j) \leq M$. Such a statement explicitly reduces the window sizes so that the two windows together can not contain more tuples than the size of the allocated memory. This is also the case for $R_i \bowtie S_j$. It has to be so that $B(R_i) \leq M - B(S_j)$ to verify that the join is deterministic; if some of the tuples in R_i were temporarily written to the hard disk when the window of S_j was in memory, the query would have given a wrong answer since some of the tuples were not available. Note that a stream and a relation have different qualities, and that we require correct results from the relation, according to the preceding DBMS presentation.

In the following is a discussion of some of the additional requirements that are presented in Section 2.3.

The DSMS literature discusses three main window designs [GÖ03]; the *sliding window*, the *jumping window*, and the *hopping/tumbling window*. Chandrasekaran et al. [CCD+03] also discuss the *landmark* window, which links between the prior discussion and the following. We start by giving a short description of this type of windows.

Landmark Window

The landmark windowing technique and the append-only database have much in common. Gehrke et al. [GKS01] defines the landmark window to have a fixed point from where the window moves. The window increases in size while tuples are added. This solution poses many of the same challenges we discussed above. One issue with the landmark windows is that they do not access historical data [CCD+03]. As we see in the following presentation, this is a common quality for all the windowing techniques, and thus a necessity to avoid blocking. The landmark

[2] and join between relations, but this is outside the scope of this thesis.

window is shown as (a) in Figure 3.6. It shows that tuples are added as filled squares as time elapses. Time is shown as the vertical arrow pointing downwards. The horizontal arrow shows how the window evolves. As we can see, the landmark window does not remove any tuples, thus the number of tuples in the window increases.

Sliding Window

The sliding window resembles the windowing technique we have discussed so far. A window has e.g. a specified time length, l and a specified time τ, so that $0 \leq l \leq \tau$. The example from [TGNO92] shows a window of five minutes, and all messages that have not been replied to will be deleted after this period. Strictly speaking, this means that a tuple that enters within the window borders stays inside the window for l time units before it is overwritten or de-allocated.

Note that there are two different types of windows. One type is *physical* windows. These use time as constraint, i.e., they as specified by e.g. five minutes, or ten seconds. The other window type is the *logical* window. This window does not use time as constraint, but rather use number of tuples to decide the window size. With the latter, the required memory is know *a priory*, and the space can be allocated at the registration of a query. The physical windows need to allocate memory dynamically, since the stream's behavior depends on fluctuations in the data arrival rate [CCD+03].

Illustration (b) in Figure 3.6 shows the sliding window. The sliding window is re-calculated for each time stamp. One consequence is that, for the window length l, a tuple that matches a condition is reported l times, given that the re-calculations are performed each time stamp. This is correct behavior due to the specification, but not always what is intended.

Jumping Window

In cases where the recalculations in the sliding window are not required, the *jumping* window offers an alternative solution. Instead of *sliding* over the stream, the jumping window fills the window with tuples and performs the calculations. After it is finished calculating, it starts to fill up a new window. This removes the re-calculation of the tuples, but does not ensure correct results, since e.g. a message can get its reply in the next window. The jumping window is illustrated in (c) in Figure 3.6. The two sets of tuples are disjoint.

Tumbling Window

The third alternative is the *hopping* or *tumbling* window, seen as (d) in Figure 3.6. If the sliding window, w_s, is re-calculated each time stamp, and a jumping window,

w_j, is re-calculated each lth time stamp, the tumbling window, w_t, can re-calculate by a time interval T such that $T(w_s) < T(w_t) < T(w_j)$.

There is also a fourth alternative that represents the cases where the update interval is larger than the window size. This causes pausing in the updates equivalent to the difference between the end of the update interval and the end of the window.

As mentioned, windows are used in blocking operators like joins. When tuples from two windows intend to join, one of the windows scans the other while blocking the input streams.

3.4.2 Approximation and Optimization

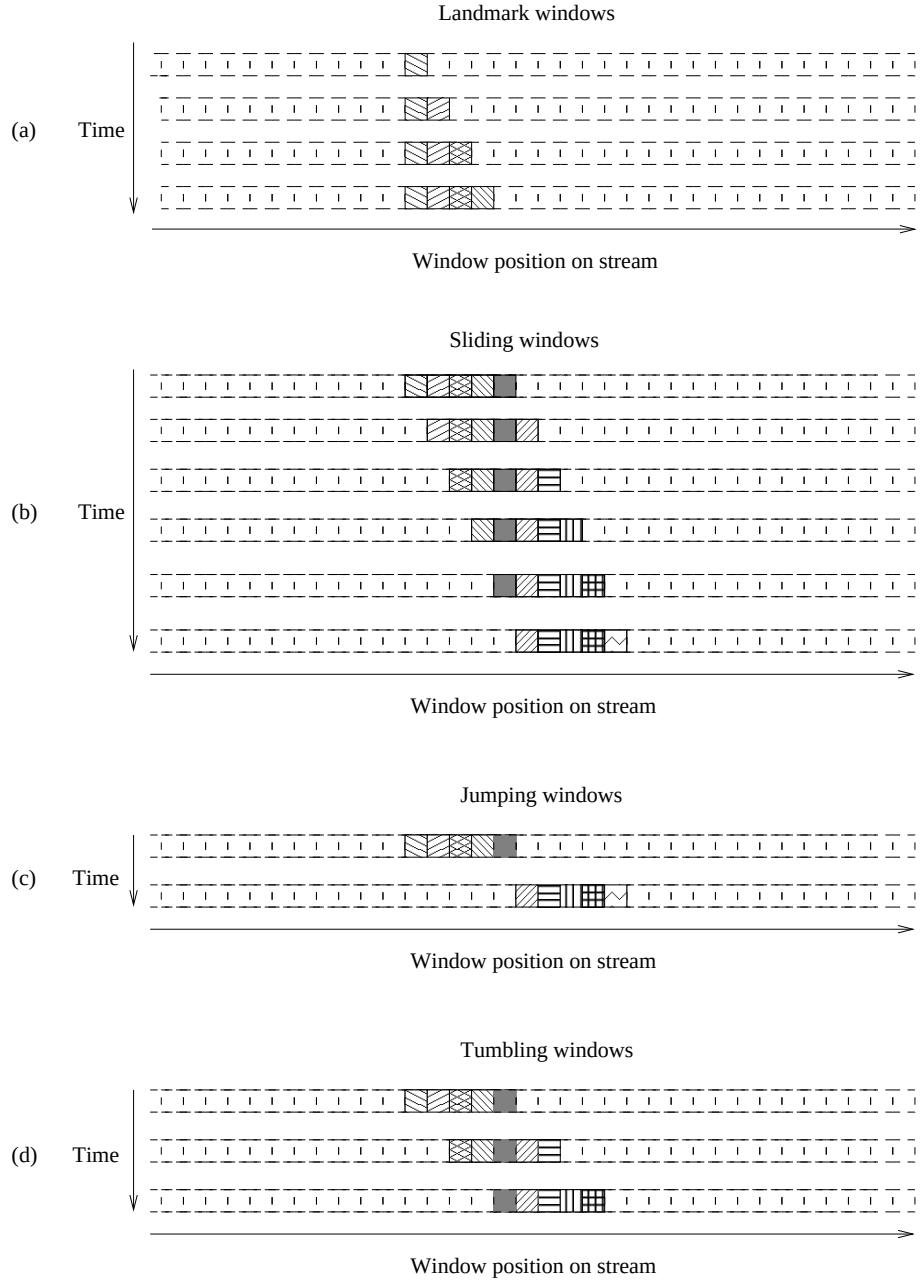
Approximation

Next to choosing the best window size semantics, the choice of window size also depends on storage capacities, speed of the system, and input rate of the data stream. If, as a worst-case scenario, the average data rate is M, i.e., the size of the memory, per time stamp τ, the maximum window size T has to be set to $T = \tau$. Such scenarios may lead to scarce system resources, since the available memory is only used for storing tuples. This also has to be taken into consideration when using several concurrent queries. If each query uses a window of size M, the system runs out of resources at once the queries need the DSMS to allocate memory.

The main solution is to reduce the number of tuples. In such cases, the DSMS has to hold some mechanisms for removing, but still registering the thrown away tuples, i.e., *shedded* tuples. Golab et al. [GÖ03] and [Loa] sum up several of the different solutions suggested in the DSMS literature:

- *Counting.* The function counts the number of tuples e.g. that match a query. This may be useful for applications where the frequency of items is important, but where the tuple's details are not in focus.
- *Hashing* uses functions to hash the dropped packets into n buckets, such that the frequency is incremented per match in the hash table. This may be helpful if the stream contains several equal tuples.
- *Sampling* reports samples of the tuples that have been shedded, e.g. values representing an average.
- *Sketches* use a random number chosen from some distribution with a known expectation to decide for the tuple's appearance. All other tuples are shedded.
- *Wavelets* are based on the same idea as sketches; the choosing of tuples is based on calculations on probability of appearance over a large set of tuples.

Figure 3.6 Sliding, jumping and tumbling windows



Optimization and Adaption

The memory consumption is a relatively important factor when handling the data streams. Queries that only filter tuples from a single stream does not require any extensive amount of memory; each tuple is compared to the filter and either ignored or sent to output.

For queries that use aggregating operators like average, or operators like join, some optimization may be required. As mentioned earlier in this chapter, a general optimizing technique is to push projections as far as possible to the source [GMUW02], i.e., the entering data stream. An example is the two relational data streams $S_1(A, B, C)$ and $S_2(C, D, E)$ and the query

$$\pi_A(S_1 \bowtie S_2) \tag{3.3}$$

may be optimized such that

$$(\pi_{A,C}(S_1)) \bowtie (\pi_{A,C}(S_2)). \tag{3.4}$$

This naive optimization technique may not always work. For example, if several queries project different attributes from the two streams S_1 and S_2, the projection has to be located longer up on the query tree.

When joining two streams that arrive in different rates, it is also important to consider re-ordering the joins, i.e., *adapting* to the stream characteristics as they change. An example is to prefer that one specific window searches another for join equalities instead of the opposite. As an alternative, the Telegraph project has proposed a solution called an *eddy* [AH00], which, instead of sending tuples up a query tree, sends them to query operators that are connected only to the eddy.

As mentioned in [GÖ03], the fact that distributing the query processors may remove some load from the main query processor, might be an optimization alternative. In sensor networks, as they are described in Chapter 2, the nodes may perform simple filtering, e.g. selections, before they send their data. In network monitoring, several routers may send their results to a final machine that performs the final calculations. The distribution of queries to separate machines may resemble optimization of the query tree.

3.4.3 Query Languages

So far, the DSMS and its functionality have been formally described. Though, there are several implementations of DSMSs. Those implementations use languages that are mainly inspired by the already existing DBMS languages. As stated in the DBMS overview, the most common query language is SQL [EM99] over relational databases. We base our discussion on SQL's declarative semantics, since SQL is the language that is most commonly used in relational modeling of data. Note that one

important aspect with DSMSs is that they use a declarative language that makes it easy for the user to understand what the query is supposed to do. This makes the code more portable, and makes it easier to verify. Today many applications are written in Perl or other high level languages, which often makes it hard for other users to understand the source code, as it tends to be complex, and not well documented. The declarative languages therefore make it easier to share knowledge. They also create a common platform for discussing, understanding, and further develop the actual query [PGB+04].

If, though, the query is aggregating, or joining several streams, it is necessary to add window semantics to unblock the query. If the current DSMS supports the possibility of joining several streams, it is necessary to specify which stream having how large windows, as well. If the systems provide support for several windowing techniques, this also has to be specified. In Chapter 4 and Chapter 5, we show examples of such queries. Here, we only show a general example of windows.

In the early versions of the TelegraphCQ DSMS the window was specified by a function `WindowIs()`. Let `S` be a stream and `ST` the start time of a query. A sliding window of five time units `t` running for 50 `t`, can be expressed as the following function [GÖ03, CCD+03]:

```
for(t=ST; t<ST+50; t++)
   WindowIs(S, t-4, t)
```

Thus changing the variables makes it possible to support all the four window types presented in this chapter. For example, a jumping window can be expressed writing `t+=5` in the `for`-loop statement.

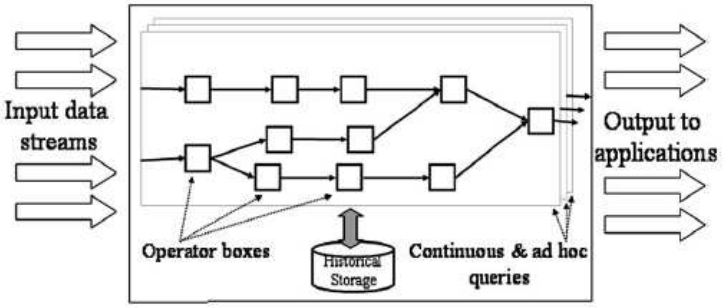
Other examples of CQ models, given in [GÖ03], are the list-based models, the time series models, and the sequence models. These models' query languages either add extensions to SQL, which makes them better suited for their applications, or they introduce alternative languages, e.g. where the user defines the streams and operators by drawing arrows and boxes, respectively. This is beyond the scope of this thesis.

As in SQL99, the need for sub-querying may also simplify queries in continuous query languages. In theory, it is possible to support sub-queries if the DSMS supports multiple queries. Though we have experienced that there are still limitations in the current CQLs with regard to sub-querying.

3.5 Examples of DSMSs

Since the data stream systems have been a hot topic the last few years [GÖ03] and several have been developed, we sum up this chapter by giving a short overview of some of the DSMS that exist today. Most of the information is taken from

Figure 3.7 An overview of the Aurora DSMS [ACC+03].



[GÖ03, GP]. We investigate the DSMSs with regard to the applications they are intended for, the input they accept, the operators they use, the windowing technique, languages, and if possible, optimization and adaption.

3.5.1 Aurora [ACC+03] and Medusa [ZSC+03]

Aurora is developed at Brown University and Brandeis University and mainly focus on querying sensor data. Though, both streams and tables[3] are used as data input. Aurora has a query algebra that is called SQuAl[4]. An operator manipulates boxes, as shown in Figure 3.7. The boxes are represented by a set of operators, which are used for e.g. filtering, sorting, mapping, aggregating, union and joining.

According to [GÖ03], Aurora also has support for fixed, landmark, and sliding windows. Aurora uses several optimization techniques like inserting projections, and combining and re-ordering boxes. During run-time, a QoS monitor investigates the streams and finds out if optimization is needed.

When Aurora is used in tight couplings between several machines, it is called Aurora*.

As Aurora is the few-node query engine, Medusa offers a solution for distributing Aurora over multiple nodes and organization networks. Medusa is developed at MIT, and focus on the inter-network challenges like efficient TCP/IP multiplexing of several connections.

3.5.2 Borealis [BBMS05]

Borealis inherits elements from both Aurora and Medusa. The system is developed as a collaboration between MIT, Brown University, and Brandeis University. It

[3] Abadi et al. [ACC+03] uses the term *static table* to describe windows on streams with unlimited size.

[4] SQuAl is short for *S*tream *Qu*ery *Al*gebra.

handles both inter node query processing and distribution of several nodes over large networks. The idea of distributing the system is that it has an incremental scalability, in case of e.g. high load spikes, and high availability, to monitor the system's health and perform fast fail-over. These parallel processing issues help Borealis act dynamically.

3.5.3 Gigascope [CJSS03]

The Gigascope DSMS is developed at AT&T to monitor network traffic at ISPs. The system is distributed such that some query operators are pushed to the routers to collect interesting information. Gigascope uses a query language called GSQL, which supports selection, join, aggregation, and *stream merge*. As an alternative to the relation based model, Gigascope operates on the data streams directly instead of transforming the data. Gigascope also tries to avoid blocking operators by assuming monotonicity and an ordering property on the joining attributes. If, for example, two joining streams, R and S, have increasing sequence numbers a, one can join $R \bowtie_{R.a<S.a} S$ by simply looking at the a values as they arrive. Since Gigascope's intention is to obtain data from simplex fiber optic lines, one needs data from several interfaces to get an overview of the stream. Thus, the stream *merge* operator is used to perform an union on the two streams before e.g. joining them. This can be considered an optimization technique, besides from re-arranging operators, which optimizes Gigascope as well. If the two streams have differing rates, there might be an overflow in the merge buffers. This is solved by inserting punctuation tuples in the stream to let go the waiting stream.

3.5.4 Niagara [CDTW00]

As both Aurora and Gigascope focus on low level data streams, Niagara supports data streams from Web-pages and Web searching. The DSMS aims to join millions of continuous queries by grouping similar queries together because several queries may share equalities. To identify similarities, Niagara's continuous query language uses an XML-like syntax to execute the multiple continuous queries. The queries are e.g. expressed as follows:

```
Where <Quotes> <Quote>
       <Symbol>INTC</>
      </> </> element_as $g
in ‘‘http://www.cs.wisc.edu/db/quotes.xml’’
construct $g
```

The query obtains tuples from `quotes.xml` and projects the values from the field `INTC`. If the equality operator is used and many queries obtain results from the same source, e.g., a stock exchange XML file, an XML table is constructed to include the equalities and destinations in the source files.

3.5.5 STREAM [ABB+04]

STREAM is developed at the Stanford University, and is a general purpose DSMS that aims to investigate resource sharing and adaptivity when for example several queries have common sub-expressions. The input stream is tuples, and the attribute data types can be `integer`, `char()`, `float`, or `byte`. STREAM supports the sliding windowing technique to unblock the data streams. It also uses a set of operators; *stream-relation*, *relation-relation*, and *relation-stream*. The stream-relation operator is using the windowing technique to map the data stream to a relation. The relation-relation operator uses SQL to operate on the tuples, which are now stored in the relation. Finally, the relation-stream operator uses three different stream definitions: The ISTREAM shows the tuples that are inserted into the stream. The DSTREAM shows the tuples that are deleted from the stream, and RSTREAM shows the relation as it is inside the window. STREAM supports sub-queries, which are expressed as SQL `VIEW`s.

3.5.6 TelegraphCQ [CCD+03]

TelegraphCQ is developed at UC Berkeley and aims to be a general purpose relational DSMS. It is written as part of the public domain DBMS PostgreSQL and inherits much of PostgreSQL's functionality. Still, TelegraphCQ offers some adaption techniques that makes it differ from PostgreSQL. This also implies that TelegraphCQ is not implemented with all the functions PostgreSQL supports. The query language used in TelegraphCQ is called *StreaQuel*, and it is similar to SQL except from the windowing semantics. The main adaptivity is created by a module called an *eddy*. This module sends and receives tuples that are processed by different operators. This poses an alternative to the static query tree that is used by other DSMSs. TelegraphCQ's windowing technique makes it possible to use sliding, jumping, and tumbling windows. As an opposite to STREAM, TelegraphCQ only manages to use one type of streams, which is created using a `CREATE STREAM` statement. Though, TelegraphCQ provides functionality for storing the stream to disk such that it later can be queried as a relation in PostgreSQL. The following chapter gives an in-depth description of TelegraphCQ.

Chapter 4

TelegraphCQ

This chapter describes the *TelegraphCQ* DSMS, which is used in the performance evaluation described in Chapter 7. Firstly, this chapter focuses on the implementation details and features that TelegraphCQ provides. Secondly, the considerations with regard to practical usage of TelegraphCQ are discussed. Some examples of usage are shown in this chapter and in the appendices. [Telb] and [KCC+03] also provide detailed descriptions of how to use the TelegraphCQ DSMS.

TelegraphCQ is developed at UC Berkeley's Computer Science Division and is part of the *Telegraph* project [Tela]. The Telegraph project is developing and analyzing different tools for monitoring data streams, and TelegraphCQ is the implementation of the Telegraph data flow engine. At first, TelegraphCQ was written in Java, but due to efforts in obtaining good performance, the system ended up being written in C [SMFH01] and based on the open source DBMS PostgreSQL [Pos]. TelegraphCQ and PostgreSQL share many of the same architectural features [KCC+03], such as user defined functions [Pos]. However, some of the features are rewritten to realize the concept of continuous queries.

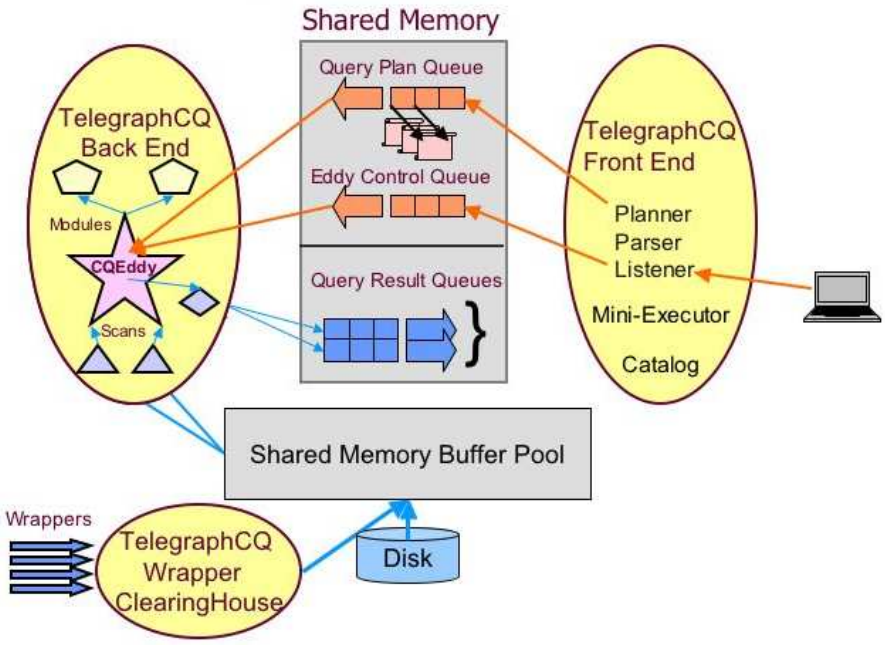
This chapter gives a detailed description of Telegraph and TelegraphCQ as they are discussed in the literature and partly implemented in version 2.0 that was released in July 2005. The terms *TelegraphCQ* and *TelegraphCQ version 2.0* are used interchangeably.

Section 4.1 gives a short description of the main concepts. These concepts are thoroughly described in Section 4.2. Section 4.3 gives a practical introduction to the usage of TelegraphCQ. Finally, Section 4.4 lists some of the limitations we have experienced in TelegraphCQ.

4.1 Architectural Overview

Most of the architectural overview is based on [CCD+03], which is the main overview paper of Telegraph and TelegraphCQ. The paper is written for TelegraphCQ

Figure 4.1 An overview of the TelegraphCQ architecture [Telb].

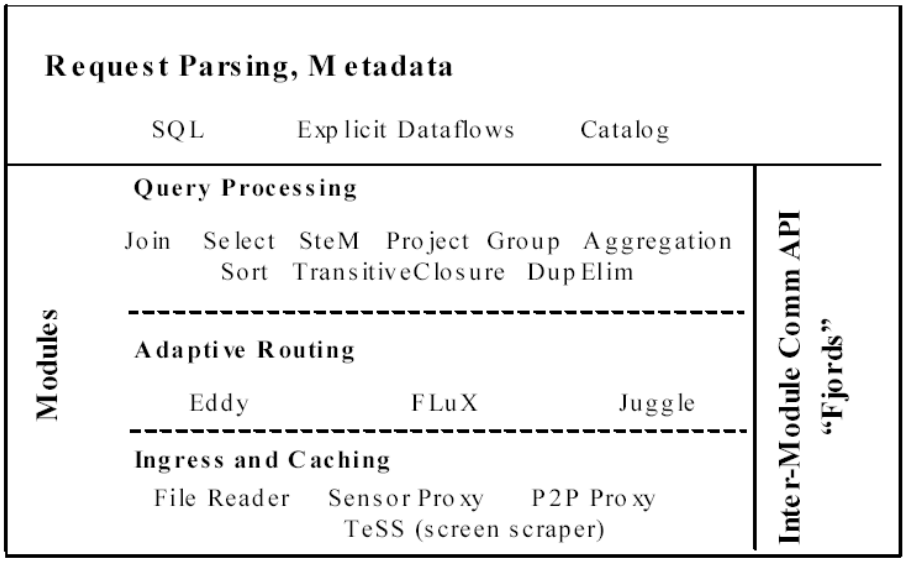


version 0.2 beta release, and prospective changes due to version 2.0 are mentioned in this chapter. This section only gives an overview over the main elements and concepts in TelegraphCQ. Section 4.2 gives an in-depth description of the most important concepts.

The main components in TelegraphCQ are shown in Figure 4.1. The two leftmost ovals are known as the *postmaster* process. The postmaster concept is adapted from PostgreSQL, and is the PostgreSQL multi-user database server. When a client contacts the postmaster, it forks a new server process that communicates with the new client. In Figure 4.1, the new server process is the rightmost oval; the TelegraphCQ *front end*. The front end parses the queries and creates the query plans as described in Chapter 3. The front end receives the result tuples from the postmaster via a *query result queue* (QRQ) as well.

As shown in Figure 4.2, the Telegraph - and hence the TelegraphCQ - architecture is based on *modules*, which are connected to each other using an interface called *Fjords*. The modules are divided into three parts, *ingress and caching*, *adaptive routing*, and *query processing*. The ingress and caching modules focus on wrapping and converting the arriving data streams to tuples that are accepted in the system. The tuples are then routed by adaptive routing modules that optimize the query execution. The adaptive routing modules then send the tuples to matching query processing modules. The Fjords interface is used to queue tuples between operators to prevent the operators from blocking. Avoiding blocking is done using queues

Figure 4.2 An overview of the Telegraph concepts [CCD+03].



that either push or pull data from the sources, as described later in this chapter.

We focus on describing the system concepts from the arrival of the data, i.e., the lower left corner in Figure 4.1 and towards the client at the front end.

The *wrappers* (Section 4.2.1) receive the data stream and turn the tuples into data types that TelegraphCQ understands. All the wrappers are controlled by the *wrapper clearing house*, which accepts connections from the external sources and sends the transformed tuples to the *shared memory buffer pool* (henceforth the *buffer pool*).

The buffer pool also provides a connection to the hard disk. Given the heritage from PostgreSQL, TelegraphCQ gives the user a possibility to join data streams with database relations, if needed.

The *TelegraphCQ back end* obtains tuples from the buffer pool and sends them to the an *eddy*[1] (Section 4.2.3), a scheduler that routes tuples to different operator modules, using a specific routing policy. These operators perform selections and joins on the tuples. When an operator is finished, it returns the tuple to the eddy, which decides whether the tuple is ready for output, or has to be sent to another operator. The eddy is an operator as well, thus an eddy can route a tuple to another eddy. The eddy operator architecture was originally implemented for single queries, and thus extended to handle continuously adaptive continuous queries (*CACQ*, Section 4.2.5) in the TelegraphCQ system. CACQ is introduced by the concept of

[1]The concept of eddies is taken from fluid dynamics, and describes the swirling of a fluid when it flows past an obstacle [edd].

SteMs (Section 4.2.4); modules that make more independent decisions about joins than the simple join operators do.

If the tuple is ready for output, it is sent to the *query result queues* in the shared memory. There it is picked up by the client who has queried for these tuples. This happens in the server-client processes that are started by the postmaster.

When a user enters a query in the *TelegraphCQ front end* client, the query is parsed and transformed into a query plan that is sent to the shared memory and picked up by the eddy. The eddy integrates the query to its operator/module concept.

4.2 Description of Concepts

The first section describes the *wrappers*, a term describing the functions handling the arrival of data. The second part discusses the TelegraphCQ back end, mentioning the *Fjords* interface, but mostly focusing on query optimization using *eddies*, *CACQ*, and *SteMs*. We also discuss the *aggregation operators and windows*, and finally, *punctuations*.

4.2.1 Wrappers

The wrappers aim to receive and obtain tuples from an external source or several external sources [KCC+03, Wra], which is a necessity for TelegraphCQ to handle real-time streams. TelegraphCQ provides both wrappers that receive - *push* - the data from the sources, and wrappers that obtain - *pull* - the data from the sources. The wrapper contains a set of functions that are defined to change the arriving tuples into the internal PostgreSQL representation of the data; the *Datum* item.

The tuples arrive via a TCP/IP socket, and the tuples are represented as e.g. comma separated values (CSVs). Changing of the tuple so that it gets the correct format is done by the user. This is solved by using e.g. a batch program that filters data from dump files, or solved real-time, by inserting a filter that transforms and sends the data to the wrapper when the data arrives. For our experiments, we have developed a filter called `fyaf`[2], which reads packets from the network interface card (NIC), changes the data into the CSV format divided into packet header fields, sends the tuples to the wrapper, reports possible packet loss, number of tuples, and for how long it received data. The user can specify which wrapper and stream the tuples are sent to, and which port the wrapper listens to. *fyaf* and its architecture is thoroughly described in Chapter 7. The TelegraphCQ distribution provides a script, `source.pl`, which supports the sending of tuples to a wrapper [Wra]. Another alternative is to use Linux programs like `sed` and `awk` to manipulate the data by using Linux pipes between them to get the correct format.

[2] *f*inal *y*et *a*nother *f*ilter. The current version is 7.

When using the *push* function, a source explicitly pushes the data to the wrapper. An example of this is network data, which cannot wait, or certain time critical sensor environments like alarms. Since this thesis focuses on network monitoring, the *push* function is used when a stream is created.

The *pull* wrapper function makes TelegraphCQ explicitly contact the source. This corresponds to the traditional DBMS view, where the system contacts e.g. the storage device for data tuples extraction. An example of such an application for TelegraphCQ pull source is to contact a specified mail server for data once a minute, e.g. to see if any replies to a message have arrived.

TelegraphCQ is constructed such that it manages to obtain data from several sources, and hence run several continuous queries over these sources simultaneously. To proceed as efficiently as possible, the wrappers are located in a *wrapper clearing house* (WCH). The WCH is the process that accepts connections from external sources and loads possible user defined wrappers and sends their definition to the TelegraphCQ back end for further tuple processing. TelegraphCQ provides an interface such that users can write their own wrappers adapted to certain formats, as well [Wra]. TelegraphCQ also comes with some built-in wrappers, e.g. a CSV wrapper, a raw packet data wrapper, and a Web page wrapper. The former can be used in the Telegraph Screen Scraper (TeSS) project for Web page querying, which is not covered in this thesis.

As mentioned, the wrappers use TCP to communicate with the external source. Due to the possibility of fragmentation, the TCP layer ensures that all the packets arrive. When the wrapper is finished building a tuple, it sends a message to the WCH, which obtains the tuple. The WCH sends the tuple to the buffer pool.

In cases where the data streams may arrive in rates that are too high for TelegraphCQ to cope with, the WCH keeps a count of all the shedded tuples, an architecture that is called *Data Triage* [RH04]. If the WCH is instructed to, the results are summarized and periodically sent to the back end as a data stream. Thus, the user can define and query the Data Triage stream to keep a count for both dropped and kept tuples [RH06]. [Loa] gives an overview of several options that are supported to give different Data Triage results.

Since TelegraphCQ is integrated with PostgreSQL, the buffer pool is also attached to the hard disk. This opens for joins between tables and streams. This also means that the streams can be defined as either `ARCHIVED` or `UNARCHIVED`. The `ARCHIVED` streams are sent to the hard disk from the WCH as well as being sent to the buffer pool. The `UNARCHIVED` streams are not stored. How the streams are defined is shown in Section 4.3.

When the number of attributes in the input streams does not correspond to the defined number of attributes, the wrapper tries to pad with `NULL`s for the missing attributes, or truncate attributes if there are too many in the arriving tuple [Wra]. This property may cause errors if the wrapper expects an `INT` but receives a `CHAR` in a certain attribute.

As we have described, the wrappers and the WCH provide functions for transforming the tuples into a format that is understood by TelegraphCQ. The user can choose which wrapper to use, and the wrapper tries to transform the data as fast and dynamically as possible.

4.2.2 Fjords

Fjords (Framework in Java for Operators on Remote Data Streams)[3]acts as the inter-module interface in the Telegraph and hence TelegraphCQ architecture. Thus, much of the communication in the back end is performed by the Fjords.

The Fjords interface stresses a non-blocking behavior on the communication between the operators. As a part of this, Fjords uses two types of connections; *push* and *pull*. This makes the Fjords support both pushing data from streams, and pulling data from relations, or a combination of the two [CCD+03, MF02].

4.2.3 Eddies

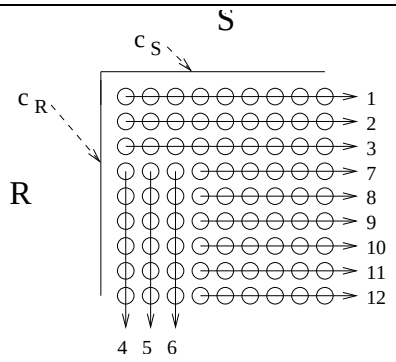
The back end process, `tcqbackend`, obtains tuples from the buffer pool. It contains several modules that aim to support TelegraphCQ in such a manner that the DSMS adapts to the streams characteristics. Figure 4.2 shows three adaptive routing modules. We focus on describing the eddy and Flux, since they are implemented in TelegraphCQ. The eddy is described in the following. Flux is described in Section 4.2.6 and we do not use this module in our experiments.

As mentioned in Section 4.1, in the Telegraph project, an *eddy* is used to dynamically adapt to changes in the data stream. This poses an alternative to the static query plan used in a traditional DBMS [AH00]. Eddies are designed for *single query processing*, but are extended to manage multiple continuous queries in TelegraphCQ. Thus we start by describing the eddy. We motivate for its design, by describing rearrangements of query trees; which may be necessary when stream characteristics change.

We start with a general example. A join is running over two *unsorted* sources R and S where e.g. $R \bowtie_a S$ is implemented by comparing a single $R.a$ with all arriving $S.a$s. When the last tuple in S is compared, a new tuple from R is compared with the $S.a$s. In other words; R is an outer loop on S. Avnur et al. [AH00] exemplifies this example by introducing two relations, `fasthi`, having high load and high values in its join attributes, and `slowlow`, representing the opposite, i.e., low load and low values in its join attributes. If `fasthi` is the outer loop, it is postponed while waiting for all the input from `slowlow`. This is called a *synchronization barrier*; `fasthi` is forced to wait - and thus block the stream - at the barrier before new

[3]As mentioned earlier, TelegraphCQ was originally implemented in Java. The term *Fjords* is probably still kept since it is allegoric.

Figure 4.3 Relations *R* and *S* as input order changes twice [AH00].



tuples can arrive. However, having parallel threads where one is forced to wait at a barrier is not considered optimal. Hence, re-ordering of the operators may obtain performance benefits [AH00]. In *nested-loop joins*, e.g. stream *S* is an outer loop joining with the inner loop of stream *R*, the optimal order for the loops changes depending on which stream is the fastest. In nested-loop joins, re-ordering can only be performed at a *moment of symmetry*, i.e., when the inner loop is finished.

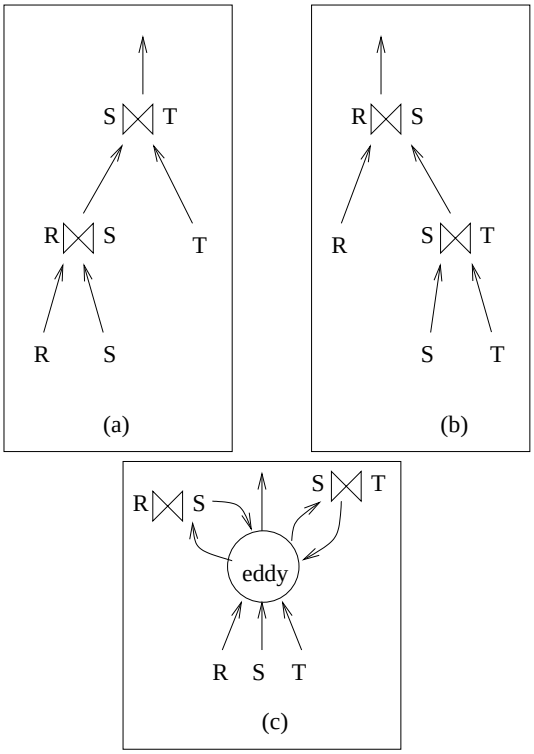
Figure 4.3 illustrates a scenario where *R* and *S* reorders twice. The small circles represent tuples that are joined between the two streams. The arrows represent the loop order. When the arrows point to the right, *S* is the inner loop, and opposite if the arrows point downwards. The numbers represent moments of symmetry. First, *S* is the inner loop. At the third moment of symmetry, *R* is set to be the inner loop. *R* starts reading from c_R while c_S keeps count of the *S* loop. At the sixth moment of symmetry, the order is changed once more, and c_S starts to read from its current position, to avoid joining tuples twice.

This principle can be extended to binary operators, as shown in Figure 4.4. Figure 4.4 (*a*) shows a join between the three sources *R*, *S*, and *T*, in which the tuples from $R \bowtie S$ are sent to $S \bowtie T$. If *R* starts to slow down, due to e.g. congestion in the network, a re-ordering might be in place, as shown in (*b*). The query tree is re-built, an operation that might take time, considered e.g. *n* binary operators and *k* streams.

As shown in Figure 4.4 (*c*), the eddy architecture is therefore proposed as a more dynamic alternative to re-building the trees. The eddy routes tuples, i.e., tuple pointers, among operators. This is a tuple-by-tuple routing, which means that it evaluates each tuple when it arrives. The operators process the tuples and send them back to the eddy. When there is no need for further processing, i.e., all operators are finished, the tuples are sent to output.

The eddy uses an array that contains two bits per operator. This array is attached to every tuple. The two bits per operator - `ready` and `done` - tells the eddy if the tuple is ready for or done with the corresponding operators, respectively. The eddy

Figure 4.4 Join trees $(R \bowtie S) \bowtie T$ (a), and $R \bowtie (S \bowtie T)$ (b) versus the eddy (c).



can therefore decide which operators are eligible for a tuple. A `done` bit is set by the eddy when the tuple is returned from the operator. If the tuple does not fit the selection criteria for the current operator, the tuple is discarded and de-allocated, and not returned to the eddy.

When all the `done` bits are set, the tuple is ready for output. By using these bits, the eddy can also force tuples through operators in certain orders, if the results depend on such static constraints. The eddy zeroes all ready bits except the one that represents the correct operator. Join operators produce new tuples from two old ones. In such a case, the `ready` and `done` bits for the two tuples are `OR`ed. In a single query environment tested on relations instead of streams, the eddy poses negligible overhead compared to the traditional PostgreSQL query processing [Des04].

To select the optimal route for a tuple, the eddy depends on a *routing policy*. The eddy keeps all tuples in a priority queue with a flexible prioritization scheme. Tuples that return from an operator are given high priority. This prevents starvation and ensures that tuples are sent to output before new tuples arrive.

In a multi-threaded shared-nothing processing framework, called a River, a *naive* eddy routed tuples using only *back-pressure* from operators or sources that did not return tuples fast enough [AH00]. The priority scheme is extended to include a *lottery scheduling* technique among the operators in [AH00]. This routing policy is implemented in the TelegraphCQ release [CVW].

Each of the operators has its own *ticket account*. For each time the eddy sends a tuple to an operator, the operator is credited by one ticket in its account. When a tuple returns from the operator, the ticket is debited from its account. Each time a tuple arrives, a *lottery* is held among the available operators. An operator that has many tickets have a higher probability of winning the lottery, thus receiving the tuple. Only the operators that have an empty in-queue are allowed to enter the lottery. This means that fast operators are favored while high selectivity operators, i.e., projection or selection operators that return many tuples, have less probability of selection, hence favoring low selectivity operators [Mou].

The problem with such an approach is that low selectivity operators may gain several tickets that cause for lottery winning even when the operator starts to slow down and other operators are better suited [AH00]. For such situations, the eddy is expanded with a *window* scheme and two types of tickets; *banked* tickets and *escrow* tickets. The banked tickets play the same role as the original tickets; they are credited to the operator when it is chosen. The escrow tickets measure the efficiency during the window time, and the banked value is replaced by the escrow value at each beginning of a window. This means that the probability increases for each returned tuple during the window, but the operator has to reprove its efficiency at the start of a new window.

Note that these windows play a different role than the window scheme presented in Chapter 3. In Chapter 3 the windows were used to reduce blocking of joins and aggregating operators. In the routing decisions, the windows are used to reduce the

eddy's probability of choosing only one operator when others may have been faster and thus more appropriate.

4.2.4 SteMs

Sometimes the solution offered by the eddy architecture is not dynamic enough. Examples of this are situations where several queries access the same tuples, or where choosing different join algorithms can be performed to gain additional adaptivity when the stream characteristics change. Taking these issues into consideration, the *state modules* (SteMs) [RDH03] simplify and extend the eddy's strength, but simultaneously increase the risk of duplicates and missing results. We show how the SteMs work and motivate for the way they especially solve joins more dynamically than the eddies.

The architecture presented in [RDH03] introduces several modules that work together with the eddy module to select, access and join tuples.

The *Selection modules* only return tuples that fit a certain selection predicate. In TelegraphCQ, the selection module is replaced by the *grouped filter*, which is described in Section 4.2.5. *Access modules* encapsulate single access methods over data sources. The access method can e.g. be a scan or an index. It receives a *probe* tuple from the eddy, and then tries to return concatenations of the probe tuple and the tuples that match. The probe tuple is returned - or *bounced back* - to the eddy in case it is needed in other modules.

The access module can be used to scan the external sources as well. In such cases, it receives an empty *seed* tuple from the eddy. The seed tuple tells the module to start sending all tuples it possesses. When all tuples are returned, either from a table probe or a scan, it sends an end-of-transmission tuple to inform the eddy to not expect more tuples from the access module. In case of general data streams, the access module scan may send data infinitely, thus an end-of-transmission tuple may never arrive. A proper close-down of the TCP socket between the source and the WCH makes the wrapper sending an end-of-transmission tuple.

The *SteM* module is similar to a half-join, and offers doubly-pipelined paths for the tuples, and each source has a corresponding SteM. Using the sources *R*, *S*, and *T*, Figure 4.5 shows how a query is traditionally represented (a), used with an eddy (b)[4], and finally, how the SteM divides the joins (c).

There are two types of tuples that interact with the SteMs. A *singleton* tuple is a tuple that arrives from the source and is still not joined. An *intermediate* tuple is a tuple that is a result from a join, but not yet ready for output. The intermediate tuples may cause for removal of more than the single ticket the eddy gives an operator, thus making the account for certain SteMs to be less than zero. As a constraint, the lower limit of tickets in the account is set to zero [Mou].

[4]We refer to Figure 4.4 in Section 4.2.3 section for additional information.

When tuples arrive at the eddy, they have three functions. We mainly focus on the first two. The first function is the probing, which is described above. The second function is to *build* tables in all the corresponding SteMs. The build tuple is added to a table in the SteM, and possible indexes are updated thereafter. When the probe tuple arrives, it returns intermediate tuples; the concatenation of the probe tuple and matching Stem tuples. The third function is optional. It is to send *eviction* tuples to the SteMs to delete certain tuples.

A logical constraint is that a tuple has to be sent to all its associated SteMs as a build tuple before it is sent to probe other SteMs. The build tuple is also tagged with a globally unique sequence number. This sequence number is used to remove duplicates. Given the query

```
SELECT
       R.a
FROM
       R, S, T
WHERE
       R.a = S.a AND R.a = T.a;
```

and a sequence number *seq*, the tuples T_R , T_S, and T_T, where $seq(T_R) > seq(T_S) > seq(T_T)$, arrive an eddy having two SteMs $R.a\triangleright$, $S.a\triangleright$, and $T.a\triangleright$. All three tuples probe their respective SteM. If T_R is sent to $S.a\triangleright$, the result tuple is valid and thus sent to output. If the eddy decides to send T_S to $R.a\triangleright$, the tuple is discarded. If a new tuple enters from R and the eddy decides to send a probe tuple from T to $R.a\triangleright$, only the youngest tuple matches thus only returning the intermediate tuple.

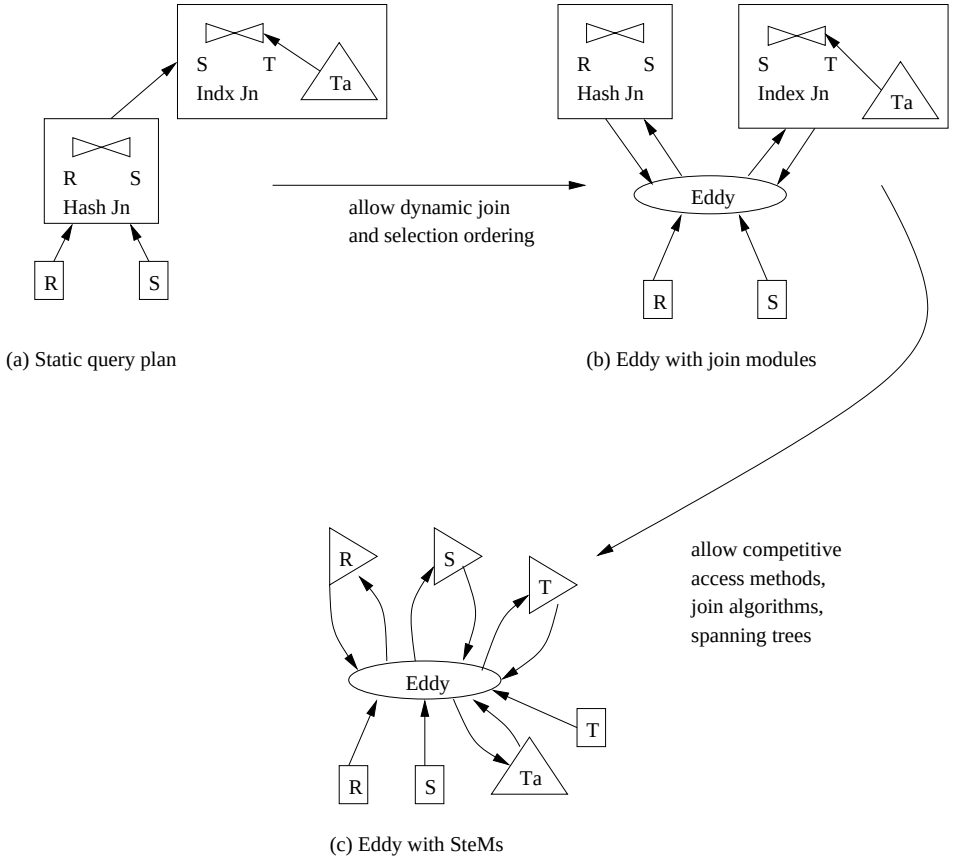
Currently, the SteMs are implemented using hash indexing. The logical consequences of such an implementations are as following:

- *Only equi-joins work properly.* Hashing a value is fast, and thus returns the matching tuples quickly. Operators like "$<$" and "$>$" may give correct results, but only on tuples having a corresponding match in the hash tables to begin with. As a consequence, we get the following point.
- *Non-equi-joins do not work.* The hashing does not give any results, since there are no matches. In the mailing list concerning the TelegraphCQ project, a work-around has been proposed by the developers. It is to insert an attribute *a* that is always true in both the streams or relations which are supposed to be joined. This results in a match in all the tuples in the SteM. The next is to add the attributes that are *not* supposed to match. This results in the following condition (using stream S and relation R):

  $$\pi_A(\sigma_{S.a=R.a \wedge S.k \neq R.k}(S \bowtie R)).$$

  As we see, this work-around results in a match on all the tuples in the SteM before the non-equalities are calculated. This may cause a significant overhead in which we only are interested in some tuples.

Figure 4.5 The development of SteMs [RDH03].



An alternative solution would have been to implement the SteMs using e.g. B-trees, which is much used in DBMSs. Though, as the structure has to be located in memory at all time, other structures, like simple arrays may also be acceptable solutions.

4.2.5 CACQ

The prior sections discuss some of the main building blocks in the Telegraph architecture. These are further extended to fit the multi-threaded environment presented in TelegraphCQ. TelegraphCQ is designed to handle several continuous queries from several clients [CCD+03]. Madden et al. [MSHR02] introduces the concept of *continuously adaptive continuous queries* (CACQ). The optimization of using multiple queries is proven to be NP-hard [SG90], thus, some heuristics have to be inserted to handle multiple simultaneous queries efficiently. CACQ aims to solve some of these challenges by adding even more information to tuples and query structures that the eddy does. We show this in the following text.

One of the main challenges of using multiple continuous queries is that the queries might have an interest in investigating the same tuples. This implies that a tuple may traverse the eddy several times before actually being sent to the query result queues. Several queries may also include the same *join* operators. CACQ has to support functionality for these queries to share the resulting tuples from these operators.

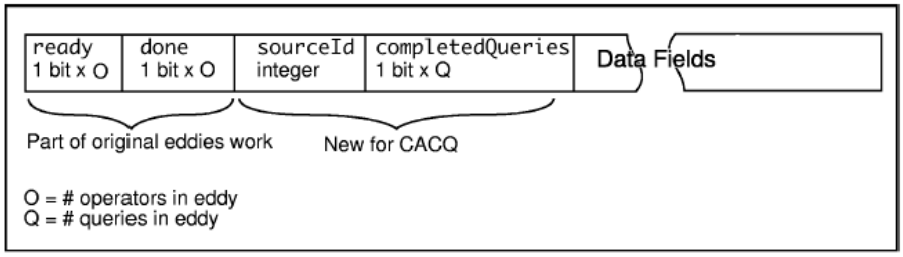
In cases of several queries and operators, the eddy has to send the tuple by looking at the tuple's *lineage* to find out which queries have used the tuple already, for example.

CACQ is also extended so that is uses *grouped filters*, which simplifies filtering by sending a tuple to a group of selection aggregates that select for that certain tuple's attribute value. Grouped filters are described later in this section.

As mentioned in Section 4.2.3, the eddies use an array of `ready` and `done` bits per tuple to describe which operators and SteMs the tuples have been visiting. CACQ extends the `ready` and `done` bits by introducing an additional `queriesCompleted` bitmap that represents each complete *query*. This bit helps identifying whether or not a tuple has been sent to and/or rejected by a query. In cases where an operator is shared by several queries, the `done` and the corresponding `queriesCompleted` bits are set. The number of concurrent queries, i.e., the size of the bitmap, is set statically by a configuration file read by the postmaster at startup or explicitly sent by the postmaster administrator during run-time[5]. This reduces the overhead by dynamically changing the bitmap on ongoing tuples when new queries are entered into the system. Of course, such a static configuration affects the choice of application domains in such a way that the domains are forced

[5]The configuration file used in the following experiments is located in the DVD-ROM.

Figure 4.6 The CACQ tuple [MSHR02].



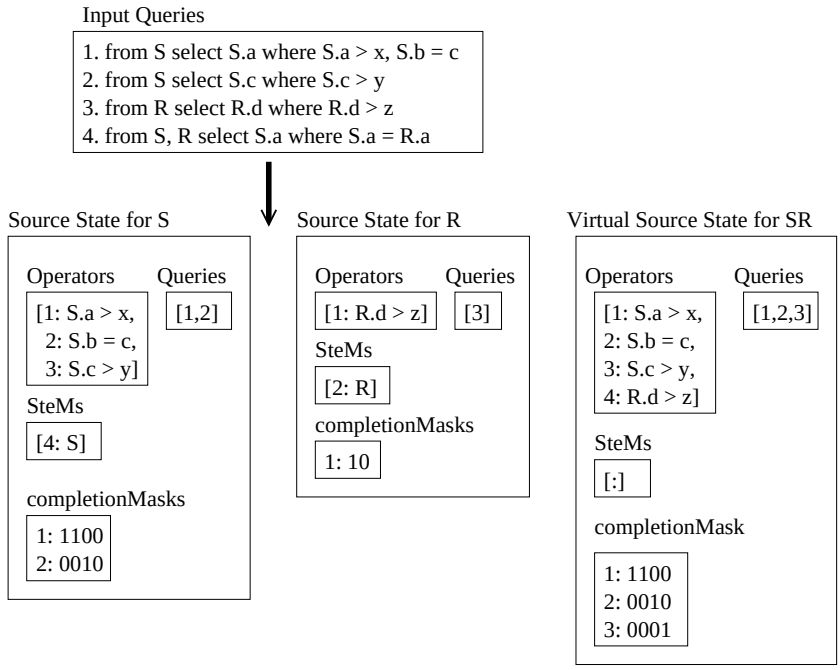
to provide some predictability with regard to e.g. number of sources and queries. For example, if one assumes ten sources and configures the system thereafter, but really receives data from one hundred sources, there will probably be some errors. Potential changes within the predefined limits are only registered at new arriving tuples. This means that when a tuple is inside CACQ, it is sent to queries that were registered when the tuple was inserted into CACQ. If a query is removed from the system, the `queriesCompleted` bit for that query is set to one, which means that no tuples are sent to the query.

The final add-on to the tuple is a `sourceId` integer, which tells the eddy which source the tuple originated from. The advantage of using this label is when queries do *not* share tuples. If for example - for n queries - query Q_1 to Q_i ask for attributes in the stream S and queries Q_{i+1} to Q_n for attributes in the stream R, there is no need for R and S's tuples to share operator and query bits. Figure 4.6 shows a tuple as it appears in CACQ. We see that when using an extensive set of l queries and k operators, the additional information per tuple is $2k + l$ bits.

In CACQ, each source has a structure associated with it, telling the eddy which operators, SteMs, and queries that need tuples from that specific source. When a tuple arrives from a source, the eddy can investigate the fields in the structure to e.g. decide where to send the tuple. Figure 4.7 show how the structure looks like in the CACQ environment. The source structure's *Query* field tells which queries using tuples from that source. In the figure, we see that query 1 and 2 only use tuples from stream S. Similarly, R only has query number 3 that only uses the tuples. The operators used by the query are split and put into the *Operators* field. The `completionMasks` field tells which operators have to be finished before the tuple is output. When the tuple returns from a source its `done` bits are `AND`ed with the source's `completionMasks` field to see if the tuple computation is completed.

When SteMs are used to join between two sources, some additional information is put into the source structure and the tuples. The tuple gets `ready` and `done` bits for the SteM and a SteMs field is added to the structure. When this is per-

Figure 4.7 The source state structures for sources. Modified from [MSHR02].



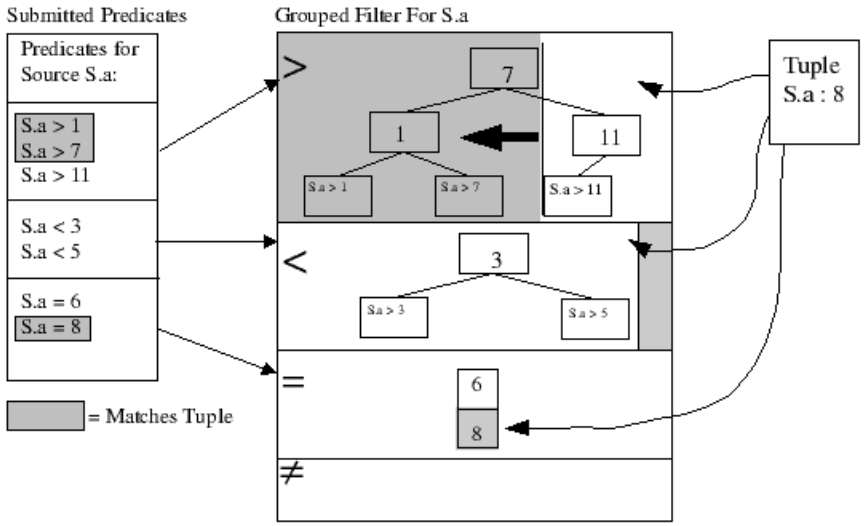
formed, the `completionMasks` field is padded with zeros to indicate that the SteM is not finished. When the tuple is sent to the corresponding SteM, the resulting intermediate tuple causes the creation of a new *virtual* source structure with an unique `sourceId`. This new structure inherits the operators and queries from the old structures, as shown at the rightmost rectangle in Figure 4.7. For k sources, there is a possibility of 2^k intermediate tuples [MSHR02]. This is the reason why not all virtual source structures are created when the system starts. The new intermediate tuple's `queriesCompleted` bitmap is cleared and gets the virtual source's `sourceId`. The `ready` and `done` bitmaps are inherited from the old tuples. When a new query is added, it is first added to the queries list in the virtual source that ranges over the sources used by the query. If the virtual source does not exist, it is created.

Note that the intermediate tuple is a new tuple, and all the probing tuples that are sent to the SteMs are bounced back to the eddy for further processing.

In the Java implementation of Telegraph with CACQ, it was experienced that the eddy's routing decisions could consume significant portions of overall execution time. This was probably caused by the tuple-by-tuple routing performed by the eddy. Chandrasekaran et al. [CCD+03] proposes usage of *batching tuples* which dynamically adjust the frequency of a tuple's change in routes. This means that lotteries are only held once in a while instead of for each tuple.

In TelegraphCQ, the CACQ environment, including the eddy and the SteMs, is

Figure 4.8 Grouped filters [MSHR02].



implemented as multi-threads [CCD+03]. The back end executor threads are collected into a single *execution object* (EO) thread. This object keeps track of the various operators and queues by acting as a scheduler between them. The EO uses a set of *dispatch unit* (DU) threads to non-preemptively schedule tuples to the operators, using a Fjords-like behavior.

Grouped Filters

As mentioned in the prior section, the CACQ implementation offers a system for organizing and grouping selection predicates[6] for multiple queries [MSHR02]. Each source has its own grouped filter that is handled by the eddy as a single operator, and each filter is divided into four data structures: a greater-than balanced binary tree ($>$), a less-than tree ($<$), an equality hash-table ($=$), and an inequality hash table ($\neq$). Figure 4.8 shows how the different predicates are grouped into the distinct filters. When the tuple from source S having value $a = 8$ arrives, all the tuple's corresponding `done` bits - in the matching sub-tree - are set. The `queriesCompleted` bitmap is set to those queries that do *not* match the predicates. This also reduces the amount of operators the tuple is sent to. Without this filtering, the tuple would have been sent to the operator $S.a > 11$ which only would discard the tuple. When a new query in an already existing source filter arrives, all the four data structures may need to be re-built.

[6] A *selection predicate* is the attribute used in a `WHERE`-clause in SQL.

Aggregation Operators and Windows

Since TelegraphCQ uses windows to prevent blocking of aggregating operators, we end this main section by discussing some of the issues associated with this. Windows and aggregates for *joins* are implemented as Fjords aggregate modules[7]. As shown in Section 4.3, we see that TelegraphCQ provides a windowing technique that supports sliding, hopping and jumping windows, and they play an important role when a query joins between streams where the tuples may have different timestamps.

When two tuples are joined, both tuples need to be in the same time partition if they are to be considered valid.

Each time the window updates, the aggregation query decides which tuples are to be displayed and which are not. To decide this, the operator distinguishes between *live* and *dead* tuples [Rei]. The live tuples are considered for being returned to the eddy, while the dead tuples are discarded. Reiss [Rei] describes an algorithm for how the aggregation node decides whether tuples are live or dead. An intermediate tuple consists of *base tuples* (e.g. singleton tuples), and the algorithm is performed each time the window updates:

1. For each base tuple in the intermediate tuple, do the following:
   (a) Compute the first and last time windows the tuple appears in (also check to see whether the tuple appears in no windows at all).
   (b) The birth time of the base tuple is at the end of the first window containing the tuple.
   (c) The death time of the base tuple is at the end of the window immediately after the last window that contains the tuple.
2. The birth time of the intermediate tuple is the maximum of the birth times of its base tuples.
3. The death time of the intermediate tuple is the minimum of those of the base tuples.

In case of situations where aggregating operators are not used, the SteMs use the tuple-by-tuple policy, which gives incorrect results since the aggregator module is not used [Rei].

When used as part of aggregations, the windows do not output results until the timestamp has reached the end of the window. For instance, if using a five minutes window, we can not expect to see any results before close to five minutes after the stream has arrived at the wrapper.

[7] The file is located in `src/backend/executor/nodeFAgg.c` in the TelegraphCQ source code.

Punctuations

TelegraphCQ supports punctuations. This means that the system explicitly creates an empty tuple for each new window to show that no results with a given timestamp occur before the given tuple [Pun].

Punctuation tuples are generated in the CSV wrapper and the Fjords aggregation node. The CSV wrapper sends empty tuples to the system each second, even if it has nothing to report. The Fjords aggregator modules generate punctuation tuples when it is finished sending tuples from a window.

The punctuation tuples can be shown in output if the user wants to. This is defined in the `postmaster.conf` file. This may ease the viewing of the results, as the output may be harder to grasp without such an explicit empty tuple.

4.2.6 Other Telegraph Features

The following features do not play an important role with regard to the performance evaluation; they provide functionality we do not explicitly use, or they provide functionality for sensor networks.

PSoup

PSoup is an extension of CACQ and used for accessing historical data and adds support for disconnected operations [CCD+03, CF03]. This means old data can be queried by new queries and new queries can access old data. A consideration might be an extension of the landmark window, which is described in the prior chapter, such that it can access historical data. Thus, PSoup allows for pull operators to access the historical data.

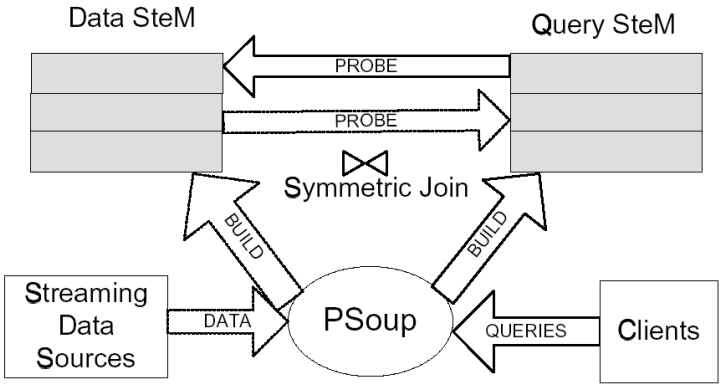
The PSoup architecture is implemented by using two types of SteMs; data SteMs and query SteMs. As data arrives PSoup, it is used to build the data SteM. The data is then symmetrically joined with the query SteM by probing for matching queries. The same procedure is followed by the queries that enter the system. They are sent to build the query SteM, and are then sent to probe the data SteM. Figure 4.9 illustrates the main building blocks in PSoup.

PSoup is only partly developed as an experimental branch of the TelegraphCQ source tree at Berkeley, and is not part of the official release.

Flux

In a shared nothing environment, e.g., in a distributed sensor network where all the sensors work independently and only share data when result tuples are sent between them, Flux is created to work as a filter between the sensors. Flux provides

Figure 4.9 PSoup overview [CF03].



two features; load balancing and fault tolerance. This means the Flux modules communicate how remote sensors may re-order their operators to change the output streams. Flux also provides a fail-over recovery, by supporting several input sensors in case one or more of them fail [CCD+03].

In TelegraphCQ, Flux uses a pre-defined port to send its data packets. In our performance evaluation, we only use one shared-all instance of TelegraphCQ, thus the Flux functionality is not used.

4.3 A Practical Overview of TelegraphCQ

The prior sections have shown some of the theoretical details in Telegraph and TelegraphCQ. This section gives an introduction to practical usage of the system. Even though there are several theoretical considerations that were discussed in the prior sections, our experiments focus on the practical usage of TelegraphCQ in a network monitoring setting.

We show how to define streams and queries, how to use the `WITH`-clause for subqueries and what limitations we experience using TelegraphCQ. Most of the information is gathered from [KCC+03], the TelegraphCQ home page [Telb], the TelegraphCQ mailing list[8], and our own experience.

Installation of TelegraphCQ is thoroughly described in both URLs in [Telb], the information gathered from both installation guides results in working instances of the DSMS.

[8]The address is `telegraphcq@yahoogroups.com`, and can be joined from [Telb].

4.3.1 Creating a Stream

We start the practical overview by showing how to create the streams in TelegraphCQ. Creating a stream is syntactically equal to creating a table in PostgreSQL. As mention earlier, TelegraphCQ's query language is mostly inherited from PostgreSQL. Though, because of the new architectural structures provided by TelegraphCQ, it is not all implemented. Only the syntax that is special for TelegraphCQ is mentioned in this section.

A stream `S` over a schema `streams` can be defined as follows (the first two commands only remove the stream and schema if they already exist and is considered to be an accepted way to remove old elements when creating new elements using SQL):

```
DROP STREAM streams.S;

DROP SCHEMA streams;

CREATE SCHEMA streams;

CREATE STREAM streams.S (
         a int,
         tcqtime TIMESTAMP TIMESTAMPCOLUMN
) TYPE ARCHIVED;

ALTER STREAM streams.S ADD WRAPPER csvwrapper;
```

`S` has two attributes, `a` and `tcqtime`. `a` is a signed integer, and `tcqtime` is a timestamp added to the tuple by the wrapper. In this example, where the `csvwrapper` is used, a timestamp is added to the tuple as it arrives at the system. This ensures a monotonic tuple ordering.

The `TYPE ARCHIVED` statement tells the system to store all the arriving tuples in such a way that they can be analyzed later using the PostgreSQL interface. If the type is set to `UNARCHIVED`, the tuples are discarded after being output or removed due to mismatch in the grouped filters or any of the modules. The advantage of storing the streams, is that one can verify results by creating queries that investigate the tuples off-line. The disadvantage is that storing possibly considerable amounts of data removes the transient impression of the stream.

As mentioned in Section 4.2.1, the usage of *load shedding* [RH04, Loa] is implemented to investigate how much tuples TelegraphCQ drops when receiving too much data. If, for example, a user wants to keep a count of the dropped tuples, it is sufficient to add `ON OVERLOAD KEEP COUNTS` after the choice of `ARCHIVED`/`UNARCHIVED` on the last line of the `CREATE STREAM` statement. The other shedding mechanisms supported are (taken from [Loa]):

- `BLOCK` stops reading tuples if the query engine is not consuming them fast enough. This is the default overload behavior.
- `DROP` means to drop Data Triage tuples without constructing any summaries.
- `KEEP REGHIST` means to build fixed-grid multidimensional histograms of shedded tuples.
- `KEEP MYHIST` means to build MHIST multidimensional histograms.
- `KEEP WAVELET` means to build wavelet-based histograms.
- `KEEP SAMPLE` means to keep a reservoir sample of the shedded tuples.

TelegraphCQ automatically creates two new streams for the shedded data; `streams.__S_kept` and `streams.__S_dropped`, which output the count of kept and dropped packets respectively.

The final line in the creation of a stream is to map the stream to a wrapper. In TelegraphCQ, the CSV wrapper is already installed, and as mentioned in 4.2.1, it changes between the ASCII format of the arriving tuples to the data format understood by TelegraphCQ.

Figure 5.1 shows an example of the streams that are used in our experiments. Section 5.1 - in the respective chapter - further discusses the creation of a stream, especially for network monitoring tasks.

4.3.2 Continuous Queries in TelegraphCQ

TelegraphCQ's query language, *StreaQuel*, is similar the the example presented in Section 3.4.3: In joining and aggregating, a *window* is specified for each stream, e.g., using the two streams `R` and `S` may give a query like this:

```
SELECT
       S.a, wtime(*)
FROM
       S [RANGE BY '10 seconds' SLIDE BY '1 second'],
       R [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
       S.a = R.a;
```

The query returns the stream `S`'s `a` attributes where `S` and `R` have the same attributes in the same time window. The window syntax uses the `RANGE BY` statement to specify the size of the window; in the example, the window lasts for ten seconds. The `SLIDE BY` statement decides how often the window is updated with a minimum of one second. Hence, the combination of these two operators makes

it possible to use sliding, hopping and jumping windows. There is also an optional `START AT` statement, which tells the system when to start the querying. If not specified, the query starts at the end of the last second before the query was submitted [Rei].

The current window semantics is implemented using the aggregate module as described in Section 4.2.5. In the projection, a built-in function, `wtime(*)` is used as well. `wtime(*)` returns the rightmost endpoint of the current time window. This means the query result shows tuples with similar timestamps if they appear in the same window. This is a practical way of e.g. investigating the number of result tuples for each window, combined with the punctuation tuples described earlier in this chapter. Sliding windows over high loads may cause significant overhead using the aggregate modules. As a final point, TelegraphCQ does not start to output tuples before the window has ranged by the given value. In the example given above, there is no output after approximately ten seconds.

4.3.3 Introspective Query Processing

TelegraphCQ supports introspective streams, i.e., streams that report internal operations during the query execution to the user [Dyn]. The current version provides three streams; `tcq_queries`, `tcq_operators`, and `tcq_queues`. The function is not yet entirely implemented and has to be manually inserted into the system as a stream definition.

`tcq_queries` gives a result from the queries that are registered during a session. The query can have entered or exited the system, thus an `E` or an `X` attribute indicates this in the stream, respectively. Each query is associated with a queue, which has a unique number. This queue is also an attribute in the stream tuple.

`tcq_operators` shows which operators have been used during the execution. The attributes show e.g. operator identification number, what queue is connected to the operator, and textual information about what kind of operator this is.

Finally, `tcq_queues` shows information about how the queue I/O has been working.

A description of the introspective streams is shown in [Dyn] and in Appendix C. Since the streams have operator and queue attributes, the actual events can be inserted into the database implicitly, by setting the stream to `ARCHIVED`, or explicitly, by performing the insertion as a batch process, and queried upon to get additional information. They can e.g. be joined on operator identification numbers.

4.3.4 Sub-Queries

In the former release of TelegraphCQ, there was no support for sub-queries. This means that earlier experiments had to emulate sub-queries by using several TelegraphCQ clients sending the result tuples from one query as a stream to another

query [GNOS05]. The problem with using such a solution is that TelegraphCQ only listens to one port, thus all communication compete on this single port. Another problem is external buffer functions. These may be given much responsibility and may affect general scheduling as well. Hence, delaying urgent tuples without letting TelegraphCQ register this and possibly shed the tuple, may happen. TelegraphCQ version 2.0 has a some support for sub-queries by allowing the user the `WITH` clause as described in [Telb] and partly in [EM99]. This means a recursive query now can be phrased as (slightly modified from [Telb]):

```
WITH
  StreamOne AS
  (
    SELECT R.i, sum(R.j) as sum, wtime(*)
    FROM R [RANGE BY '10 seconds' SLIDE BY '1 seconds']
  )
  StreamTwo AS
  (
    SELECT S.k, sum(S.l) as sum, wtime(*)
    FROM S [RANGE BY '10 seconds' SLIDE BY '10 seconds']
  )
(
SELECT *
FROM StreamOne S1, StreamTwo S2
WHERE S1.i = S2.k
);
```

The query has two sub-queries that give results to `StreamOne` and `StreamTwo`. `StreamOne` uses a sliding window that lasts for ten seconds and is updated every second. `StreamTwo` uses a jumping window that updates each tenth second. The lower query joins the two streams on the `i` and `k` attributes. As we can see, the query does not use any windows. This query adds all the tuples to the corresponding SteMs, but does not delete them. This means when a probing tuple from e.g. `S1` is sent to `S2`'s SteM, it is joined with all the matching tuples in the SteM. This corresponds to the append-only blocking behavior that is described in Chapter 3, and can also resemble the landmark windowing technique we described in the same chapter.

When using the `WITH` clause, the new streams have to be defined using the `CREATE STREAM` statement. For the example above, both `StreamOne` and `StreamTwo` has to be defined explicitly.

Sub-queries are helpful to solve tasks that are slightly more complicated than e.g. joining between two streams or project some of the attributes from the tuples. In the following chapter, we see several tasks that depend on sub-queries to return correct results. Without sub-queries, the range of solvable tasks is limited.

Consequently, there is still no support for sub-queries in the `SELECT`-clause, or for example by using the `IN` statement in the `WHERE`-clause. Though, dissecting these issues shows that alternative queries may solve the latter problem. Given for example the following query:

```
SELECT COUNT(*)
FROM R
[range by '10 seconds' slide by '10 seconds']
WHERE R.i IN (SELECT i
              FROM Q
              [range by '10 seconds' slide by '10 seconds']
              WHERE i < 42);
```

This query may be rewritten to

```
WITH
  S AS
  (
    SELECT i
    FROM Q
    WHERE i < 42
  )
(
SELECT COUNT(*)
FROM R [range by '10 seconds' slide by '10 seconds'],
     S [range by '10 seconds' slide by '10 seconds']
WHERE
     R.i = S.i;
```

4.3.5 User Defined Functions

One of the strengths in PostgreSQL is the possibility of inserting user defined functions. This makes it possible to implement the functionality that is needed. For example, Reiss et al. [RH06] use this functionality to implement set operators like union, which we have experienced is not working in TelegraphCQ, as shown in Appendix B.3. Since these user defined functions are highly dynamic, but are not part of the declarative language, we choose not to focus on this in this thesis.

4.4 Limitations in TelegraphCQ

During our work with TelegraphCQ, we have experienced some limitations that are either not documented, or not easily extracted from the literature:

- *The eddy does not understand the `OR` operator*. As shown in the following chapter, sometimes the `OR` operator might be helpful to describe selections. Madden et al. [MSHR02] describes how this feature is implemented in their CACQ system. The challenge of using `OR` is that predicates not matching a disjunct can not be discarded immediately. This is solved by adding a bit per disjunct with each tuple. When any predicate of the disjunct evaluates to true, the bit is set. The release of TelegraphCQ, which is used in our performance evaluation, is only an experimental version, thus not all features are implemented, forcing us to either find alternative solutions or describe queries in a theoretical perspective.

- In our lab setup, as described in Chapter 7, only six `AND` operators are allowed in the selection between two SteMs. If the number is exceeded, the postmaster process is shut down. Other setups have given different numbers of the operator, leading us to conclude this may be a memory issue.

- TelegraphCQ's back end stops once in a while, complaining about problems with shared memory. The log files sometimes show a mis-alignment in the tuples, or that tuples are dropped from the wrong summary window. The last error may be caused by the shedding mechanism. Since TelegraphCQ is developed and tested on the Fedora Core 1 Linux distribution, we may assume our instance running on a SuSE distribution may be the error source. But, we have experienced the same problems on a Fedora VMWARE installation, indicating that the distribution may not be the cause of the stopping. Another issue may be the version of the compiler, but we have assumed the compiler used is the same as is included in the distribution.

Chapter 5

Design and Implementation

This chapter discusses the design of the network monitoring tasks. Firstly, the IPv4 and TCP stream definition as we have expressed it in TelegraphCQ is presented and discussed. Then we present a set of tasks which investigate some of the requirements presented in Chapter 2.3. The tasks are based on solving some of the application domains presented for network monitoring in Chapter 2.

In the presentation, the tasks are mainly discussed with regard to these two issues:

- **Description** of the tasks and what they are meant to achieve. The section also discusses which parts of the streams may be of interest and what the task requires from the DSMS.

  Some of the tasks may not have any relevance outside the experimental context, but as this thesis focuses on performance evaluation, we have created tasks which investigates different features and requirements of the DSMS.

- **Query and Analysis.** First of all, this section is dedicated to the query implementation. The section shows and discusses how the query is modeled in TelegraphCQ to optimally match, and function, in the DSMS.

The first task is we design is a preliminary task. It is a simple projection task which shows how to filter all the tuples in a stream. It is used as an introduction and only shows how the DSMS can be used to store all the data packets it receives.

We have designed three tasks that are later used in the performance evaluation. Two of the tasks origin from Plagemann et al. [PGB+04]; a performance evaluation of TelegraphCQ version 0.2 which were a part of a project at the Institut Eurécom [Eur]. We discuss these tasks and argue for certain modifications of their implementation, e.g., to fit the new windowing semantics in TelegraphCQ version 2.0. The first task is simple and investigates the projection and aggregation of a stream. The second task looks at the performance of a join between a table and a stream.

Finally, the third task shows the complexity of identifying connections in a TCP stream using TelegraphCQ.

The final part of the chapter gives examples of other tasks that may be interesting for future work. Some of the tasks do not work properly in the current version of TelegraphCQ, and some are almost equivalent to the three tasks used in the performance evaluation. Each task is discussed separately with regard to these issues. The results from these tasks are not as thorough as the first tasks, and they are mostly referred to in Appendix H, which again shows where the tasks and results are located in the attached DVD-ROM.

5.1 The IP/TCP Stream

This section discusses the TCP/IP stream and the attempts to make it fit TelegraphCQ as optimally as possible. As shown in Chapter 4, TelegraphCQ's definition of streams is syntactically almost equivalent to the SQL99 table creation, except from timestamp fields and adding to a wrapper.

We have chosen to implement *all* the header fields in both the IP and TCP headers. The packet headers, as they are defined in their respective RFCs, are located in Appendix J. This means that we end up using a considerable tuple having 29 attributes. This is done by purpose. As stated in Chapter 4, TelegraphCQ claims to support many concurrent users and different queries. In the network used for the performance evaluation, it is only TCP traffic. Thus, it is a possibility of queries that use a variety of the different header fields. Thus, we can not reduce the number of tuples to only contain IP addresses and TCP ports, for instance. The advantage of using such a large tuple is that all queries in the IP/TCP stream can be tested. The disadvantage is that it probably requires much resources.

Since the tuple is large, we have tried to be pragmatic in the selection of data types; if TelegraphCQ provides a one to one support of a data type, e.g., the IP address, as shown in the following discussion, we choose to use this data type for compatibility reasons. The other solutions, where TelegraphCQ does not support the current header field explicitly - as is the case for the vast majority of the fields - are discussed more complementary.

One challenge is the difference between hexadecimal and decimal representation of packet header fields. The filtering tool, `fyaf`, which is discussed in Chapter 7, receives the raw data packets from the NIC. For us, it is tempting to continue using this representation in TelegraphCQ, for simplicity and consistence. However, the drawback is that TelegraphCQ has no `hex` data type support, which means we have to use the `text`, `char` or `varchar` representation of the hexadecimal values as replacements. Hence, the size required will be a little more than the double of the space taken per sign. For example, `0xa` occupies 4 bits, while the character `a` needs 8 bits. This representation also requires 8 bits to null terminate the string.

The alternative is to let `fyaf` change these values so that they are represented as decimal numbers. This eases the calculation, especially of the sequence numbers and header checksums. Thus, we transform some fields into decimal representation. The problem is that such a representation has to be calculated outside the DSMS, in `fyaf` or in an appropriate wrapper, since TelegraphCQ does not yet have an option for such calculations. Tests have shown that such an operation does not add any significant overhead to `fyaf`'s performance.

The first header fields that need to be discussed further - with regard to hexadecimal representation - are the IP header's `tos` (Type of Service), `id` (Identification) and `protocol`, and the TCP header's `reserved` and `urgPtr` fields. They are only viewed as identifiers. Therefore, it would have been sufficient to represent them as the hexadecimal values. This especially applies to the `protocol` field, which is used by the IP parser to identify the next header. Though, to be consistent with the protocol identification, we have chosen to represent them as hexadecimal values. Changing `fyaf` to represent only these fields as hexadecimal values is not considered an issue with regard to time consumption.

We use three different data types to identify the decimal fields; `smallint`, `int` and `bigint`. As in PostgreSQL, `smallint` is signed 16 bits, `int` is signed 32 bits and `bigint` is signed 64 bits [Pos]. `int` is the preferred data type in TelegraphCQ, and the two other types are only recommended if memory size has to be taken into consideration. Since the definition we have designed tries to satisfy both the speed requirements and the memory usage as optimally as possible, we use a simple algorithm to choose an appropriate data type:

> All the fields having less than 16 bits are set to `smallint`, all the fields ranging from 16 to 31 bits are set to `int`, and all the remaining fields, which are not bound to be represented with other data types, are set to `bigint`.

Since the data types are signed, e.g. `int` can not address 2^{32} with only positive values, we use the `bigint` to represent the 32 bits fields. This is actually a considerable waste of space; we only need one more bit to represent the value. Thus, the algorithm may not distribute optimally, but it is simple and gives a satisfying collection of the three data types.

The stream is displayed in Figure 5.1. It corresponds to the IPv4 and the TCP header as described in Appendix J. Note that the line numbers are included in the figure. Since we try to use TelegraphCQ's built-in data types for representing certain fields, the following is a description of some of the attributes that are exceptions from the data type choosing algorithm mentioned above.

The source and destination IP addresses are formatted as the TelegraphCQ `cidr` data type, which is based on the classless inter domain routing trend [RL93]. The attribute is displayed as an IPv4 address followed by a subnet indicator, e.g. `192.168.1.1/24`.`cidr` and `inet` are both used to visualize Internet addresses

Figure 5.1 The stream as it is defined in TelegraphCQ

```
DROP STREAM streams.iptcp;

CREATE STREAM streams.iptcp (
-- Explicit timestamp
--      timest int,
-- IP
        version smallint,
        ipHeaderLength smallint,
        tos smallint,
        totalLength int,
        id int,
        flags smallint,
        fragOffset smallint,
        ttl smallint,
        protocol smallint,
        headChksum int,
        sourceIP cidr,
        destIP cidr,
        IPoptions text,
-- TCP
        sourcePort int,
        destPort int,
        seqNum bigint,
        ackNum bigint,
        tcpHeaderLength smallint,
        reserved smallint,
        URG char(1),
        ACK char(1),
        PSH char(1),
        RST char(1),
        SYN char(1),
        FIN char(1),
        windowSize int,
        chksum int,
        urgPtr int,
        TCPoptions text,
        tcqtime timestamp TIMESTAMPCOLUMN
) TYPE UNARCHIVED ON OVERLOAD KEEP COUNTS;

ALTER STREAM streams.iptcp ADD WRAPPER csvwrapper;
```

in TelegraphCQ. They are both defined in the same structure, `inet_struct`, which occupies 56 bit of memory when allocated[1].

Both the *options* attributes, i.e., `IPoptions` and `TCPoptions`, are defined as `text`. This is not optimal with regard to space utilization, since each sign occupies 1 byte. We choose this solution since the option fields have varying sizes; the number of 4 bytes option fields can be zero to 10. The maximum number of characters is 81. As mentioned in Chapter 3 and Chapter 4, the number of attributes has to be constant. Hence, using the `text` data type is considered the best available solution, though using `char(81)` would also have been a good choice. This also shows that TelegraphCQ fails to meet the requirement of dynamically allocation of option fields and extension headers, as described in Chapter 2.

Finally, the control bits are represented as `char(1)`, which is the smallest possible representation in TelegraphCQ, given that we want to place all the control bits in one attribute each. Another solution would have been to represent the six bits as one integer each, or represent all the flags as numbers, e.g., `010010` would have been represented as 18. This would have led to a trade-off, since the number would require further computations to see which bits are set. Since we want TelegraphCQ to look as manageable as possible to the user and be consistent to the protocol's definition, a six bit representation is not implemented.

The stream in Figure 5.1 differs from the stream defined in [PGB+04]. The new stream attribute takes approximately 800-900 bits. A comparison shows that the new stream takes about 200 bits less space than the one in the article. Given a considerable rate of packets per second, this has some effect on the memory usage of the TelegraphCQ, and also the speed, since the article's stream uses the `bigint` data type where only for example an `int` or an `smallint` would have been adequate.

Finally, remember that TelegraphCQ adds `ready`, `done`, and CACQ bitmaps to each tuple. These static values also represent a significant overhead when the number of tuples increase.

5.2 Tasks for the Performance Evaluation

We have chosen to run three tasks for the performance evaluation. The first task aims to investigate aggregation functionality. The second task focuses on joining between streams and relations, and how concurrent separate queries affect the performance of TelegraphCQ. Finally, the third tasks looks at complex queries where several sub-queries depend on each other and work together to give results. We also include a simple preliminary task that projects all the tuples obtained by the DSMS.

[1] See `include/utils/inet.h` in the TelegraphCQ source code for details.

5.2.1 Preliminary Task: Select all packets that arrive at the DSMS

Description

This task is simply selecting all the packets that arrive the DSMS. The purpose of using such a query is to store the results into the database for later analysis. As this is not one of the intentions using a DSMS, the task is not that relevant in itself. Though, it shows that TelegraphCQ supports projections, a very important requirement for DSMSs. This task is also interesting for evaluating the accuracy of TelegraphCQ in the network monitoring application.

Query and Analysis

As we manage to project the attributes, the following query acts as a template for all the projection queries:

```
SELECT
       *
FROM
       streams.iptcp;
```

Since there are no aggregations or joins, we do not use windows. The * projection can be replaced with `destIP, destPort, sourceIP, sourcePort` to select unique connections, for example.

5.2.2 Task 1: Measure the average load of packets and network load per second over a one minute interval

Description

This task has to display two results simultaneously. Firstly, the average network load, i.e., how many bits or bytes have been registered by the DSMS each second during the last minute, has to be measured by one statement. Secondly, the count of number of packets over the same time interval has to be measured by another statement. In this discussion, we show two different approaches to a possible solution. This is done to show that there can be several alternative solutions to one task. The different solutions are evaluated in Chapter 7 to measure possible difference.

Query and Analysis

Firstly, we have chosen to use a sub-query that sends results every second, and a main query that registers all seconds and calculates the average for each minute. Figure 5.2 shows how the query is implemented. This implementation is called

Figure 5.2 The first query that calculates the average network load for each minute.

```
WITH
        streams.task1_1 AS
        (
        SELECT
                COUNT(*), SUM(totalLength), wtime(*)
        FROM
                streams.iptcp
                [RANGE BY '1 second' SLIDE BY '1 second']
        )
                                10
(SELECT
        AVG(totalNum), AVG(totalLength)*8
FROM
        streams.task1_1
        [RANGE BY '1 minute' SLIDE BY '1 second']);
```

`task_1.1`. The `COUNT(*)` in `streams.task1_1` counts the number of all packets, i.e., tuples that arrive. `SUM(totalLength)` sums up the `totalLength` attribute for each tuple. Note that this header field does not include the Ethernet header size. `streams.task1` runs a jumping window that updates each second. The main query - as shown in the bottom of the figure - receives the result from `streams.task1_1` each second, and uses a sliding window that lasts for one minute. Each time the window updates, the `AVG(totalNum)`, i.e., the first attribute in `streams.task1_1`, as shown in Appendix E, is averaged over the 60 tuples that are in the window. `AVG(totalLength)` multiplies with eight to get the average number of bits per second, since the `totalLength` field in the IP header denotes the total header length in bytes [Pos81a].

This approach might be simplified by using only one query. The motivation for using two queries is that it may simplify the understanding of the query. Though, the sub-query can be avoided. Figure 5.3 shows a one-query solution, and is called `task_1.2`.

For each second, after gathering data for one minute, the count of tuples is divided by 60 to show an average per second. The average of total length is thus calculated by simply using the `AVG` function. Note that we have chosen not to multiply with eight in this query, neither multiplying with the corresponding number of packets. This gives another result, but is easily calculated to fit the first query. We assume the overhead is minimal, but we take this into consideration when we discuss the results in Chapter 7.

Figure 5.3 The second query that calculates the average network load for each second.

```
SELECT
        COUNT(*)/60, AVG(s.totalLength)
FROM
        streams.iptcp AS s
        [RANGE BY '1 minute' SLIDE BY '1 second'];
```

5.2.3 Task 2: How many packets have been sent to certain ports during the last five minutes?

Description

This task needs a functionality to compute a new result for each arriving tuple. This gives a real-time analysis of the stream characteristics over the last five minutes. Plagemann et al. [PGB+04] use this task for joining between a table and a stream. This means they also created a table that could contain all the 65536 (2^{16}) possible port numbers, if required.

Since TelegraphCQ uses hash indexing in the SteMs, results are supposed to be calculated relatively fast.

Query and Analysis

The definition of the query is shown in Figure 5.4. It only differs from the article's query by the new windowing semantics. The query uses a sliding window that is updated each second and joins the projected attributes with the table's `port` attribute. The `WHERE`-clause could have been removed to avoid the join. If the table contains all the ports, the join is redundant. If only some ports are interesting, this can be specified in the table, or with the `WHERE`-clause as described below:

```
[...]
WHERE streams.ip_tcp.desPort = 80
      AND (streams.ip_tcp.desPort >= 100
      AND streams.ip_tcp.desPort < 200)
[...]
```

Given a significant number of interesting ports, this may affect the simplicity of the query. Internet assigned numbers authority (IANA) [ian], has defined three ranges of TCP ports that are used for certain purposes[2]:

1. *The well known* (dedicated) *ports* are those from 0 through 1,023.

[2] These are located in `http://www.iana.org/assignments/port-numbers`

Figure 5.4 The Task 2 base query.

```
-- name: task2_1.sql

--DROP TABLE ports;

--CREATE TABLE ports (
--      port int,
--      counter int
--);

SELECT                                10
      wtime(*), streams.iptcp.destPort, COUNT(*)
FROM
      streams.iptcp
      [RANGE BY '5 minutes' SLIDE BY '1 second'],
      ports
WHERE
      ports.port = streams.iptcp.destPort
GROUP BY
      streams.iptcp.destPort;
```

2. *The registered ports* are those from 1,024 through 49,151.

3. *The dynamic and/or private ports* are those from 49,152 through 65,535.

The task may be interesting for logging the traffic on certain ports, e.g., port 80. If all the ports are monitored, there is a possibility of obtaining statistical information of the distribution of used ports. The problem is that real-time analysis of, in the worst case 65,536 ports, may loose some of its intentions; it would require additional resources to monitor all the ports and attain useful information at the same time. On the other hand, the usage of three explicit tables mapped to the corresponding ranges as described above, may be another solution. If the DSMS monitors a Web server, perhaps only the first 1,024 ports are interesting for evaluation.

Since we use a sliding window, the memory stores five minutes of packets, and the `SELECT`- clause only projects the destination port fields from the packets.

The following calculation shows how much space a five minutes stream may need. `streams.ip_tcp.desPort` occupies 32 bits. With a maximum packet size of 1,500 bytes (12,000 bit), a bit rate of 100 Mbits/s (104,857,600 bits/s), and a constant data stream, the DSMS will receive an average of 8,738.1333 packets each second. 32 bits * 8,738.1333 packets/s = 279,620.27 bits/s. This is 349,525.33 bytes each 10th second. One minute gives 2,097,152 bytes. Five minutes give 10,485,760 bytes, which is 10 Mbytes. To complete the size estimation, we also

have to add the table, which with its maximum number of ports occupies approximately 262,144 bytes, which is 0.25 Mbytes. Therefore, in the worst case, the query data will occupy approximately 11 Mbytes of main memory.

Since only the number of destination ports are projected, there might be a problem of further optimizing the article's query. As noted above, the maximum number of registered destination ports can be 8,739 each second at a 100 Mbits/s link. Thus, summing up the packets in one query and sending them to the next, using the `WITH`-clause will not have any effect. The task does not require any additional manipulation of the stream except for projecting the destination ports, and the DSMS has to calculate the whole window each second anyway.

5.2.4 Task 3: How many bytes have been exchanged on each connection during the last ten seconds?

Description

This task has to calculate the size of each packet's payload as it arrives at the DSMS. By *payload*, we mean TCP's *data field*. The total size of the packet, except the Ethernet header, is given in number of bytes and is located in the IPv4 header as the `totalLength` field. Each of the two headers have their own fields giving the header length. Since we are interested in the number of bytes, `totalLength - (ipHeaderLength*4)- (tcpHeaderLength*4)` is sufficient to calculate the payload. We multiply both the `ipHeaderLength` and the `tcpHeaderLength` by four, since they indicate the numbers in 32 bits words, hence four bytes.

A connection is uniquely identified by the IP address and the TCP port at both sides. In the traditional DBMSs, these attributes would have been referred to as the *unique key* in a relation holding several connections. Over a time period is can be several sequential connections on the same unique set of IP addresses and TCP ports. Therefore, we suggest that the implementation of a query, which takes these two issues into consideration requires more complex queries than in the previous tasks. We also argue that TelegraphCQ as it is implemented in the public release of version 2.0 will not manage to solve this task properly. We build up an understanding of the complexity involved in such queries, and try to use the queries as well as possible into TelegraphCQ's query language.

A TCP connection initialization is described as a 3-way handshake. Firstly, the initiator - henceforth the client - sends a SYN packet, i.e., a TCP packet with the SYN field set to 1. This is called an *active open*. Then the responder - henceforth the server - sends a packet with both the SYN and the ACK fields set to 1 - henceforth a SYN/ACK packet - which is called a *passive open*. Finally, the client sends an ACK packet. The TCP packet uses a sequence number and an acknowledgment number during the 3-way handshake, to make sure they agree on the

connection establishment. The client chooses an initial sequence number, which is acknowledged by the server, while the server sends a new sequence number in the SYN/ACK packet. The ACK packet from the client increases its acknowledgement number of the server's sequence number. This is called *forward acknowledgement*.

The description is a simplification of the protocol's actual behavior. Postel [Pos81a] defines other conducts, especially at the connection tear down, since there is a possibility of packet loss. Packet loss may lead to misconceptions between the client and the server about whether or not a connection is up. But, for simplicity, we choose to ignore these issues in this design. The most important issue is to identify the connections that are established during the time the task is performed. Thus, we discuss connection tear-down in Task 5.

Query and Analysis

Plagemann et al. [PGB+04] wrote a task that is similar to this one. The task is further simplified with regard to connection identification and is based on the heuristic that a connection does not last more than one minute. We try to investigate whether TelegraphCQ manages to handle a more detailed and protocol dependent query, as required in Chapter 2.

In retrieving more connection details from the stream, the query has to be split into sub-queries, which are responsible for the different stages of the connection establishment. A final join on the streams produces the results we are looking for.

As noted in Chapter 4, TelegraphCQ version 2.0 still has some limitations with regard to sub-querying. We show these limitations by referring to the TelegraphCQ chapter and how they affect the query design.

Ideally, the connection identification should have been performed in one operation, i.e., one simple query using a set of intuitive operators. Unfortunately, the SQL99 standard, is not constructed with online network monitoring in mind. Neither does the current TelegraphCQ implementation support the complete SQL99 standard.

Finding all the packets that play a role in the 3-way handshake could have been performed using a query that selected all the packets satisfying the conditions in either of the three handshakes. An example would have been (in pseudo code):

```
SELECT
        <attributes>
FROM
        <stream>
WHERE
        <stream>.SYN = 1 AND <stream>.ACK = 0
        OR
        <stream>.SYN = 1 AND <stream>.ACK = 1
        OR
```

```
<stream>.SYN = 0 AND <stream>.ACK = 1.
```

Since TelegraphCQ does not support the `OR` operator, the three conditions would have had to be changed to fit the logic, i.e., using de Morgan's laws, which manages to change between `OR` and `AND` using the `NOT` operator[3]. However, TelegraphCQ does *not* support `NOT` followed by parentheses, e.g.,

```
[...]
NOT ( NOT ( <stream>.SYN = 1 AND <stream>.ACK = 0 )
      AND
      NOT ( <stream>.SYN = 1 AND <stream>.ACK = 1 )
      AND
      NOT ( <stream>.SYN = 0 AND <stream>.ACK = 1 ) )
[...].
```

This would have been a plausible alternative of representing the conditions. To implement the 3-way handshake, we therefore use at least two different queries, each responsible for each of the packets identifying a connection. Figure 5.5 shows the two queries as they have evolved and are used in the task. Query (a) selects the SYN packets, while query (b) selects the SYN/ACK packets. We have chosen to integrate the identification of the ACK packets in another query, since there is a considerable amount of ACK packets in the network. We further argue for the choice of integrating the ACK packets in a later query in a following discussion. Finally, in these queries, we also implicitly assume that the other flags are set to `'0'`, though not explicitly queried for.

The results from the sub-queries have to be computed in yet other queries. We identify a connection by the initiator, which means that if the client starts the communication, the connection is identified by a tuple having this structure:

```
<client IP>,<server IP>,<client port>,<server port>.
```

Thus, the opposite combination, where the responder is located in front of the tuple, is undesired and indicates an error in one of the queries. To represent the connections correctly, the query therefore has to switch the responder's packet. It also needs to look at the sequence and acknowledgment numbers so that it captures the three packets as intended. To begin with, we suggest a `WHERE`-clause having these conditions (*syn* are the packets containing the SYN packets, *synack* the SYN/ACK packets, and - for simplicity - *ack* the ACK packets):

```
[...]
syn.sourceIP = synack.destIP
```

[3] de Morgans laws states that `NOT A AND NOT B` ≡ `NOT (A OR B)` and that `NOT (A AND B)` ≡ `NOT A OR NOT B`.

Figure 5.5 The first two queries for the connection identification

```
-- (a)

(SELECT
        sourceIP, destIP, sourcePort, destPort,
        seqNum, ackNum, tcqtime
FROM
        streams.iptcp
WHERE
        SYN = '1' AND ACK = '0')

-- (b)                                    10

(SELECT
        sourceIP, destIP, sourcePort, destPort,
        seqNum, ackNum, tcqtime
FROM
        streams.iptcp
WHERE
        SYN = '1' AND ACK = '1')
```

```
AND
syn.sourceIP = ack.sourceIP
AND
syn.destIP = synack.sourceIP
AND
syn.destIP = ack.destIP
AND
syn.sourcePort = synack.destPort
AND
syn.sourcePort = ack.sourcePort
AND
syn.destPort = synack.sourcePort
AND
syn.destPort = ack.destPort
AND
syn.seqNum + 1 = synack.ackNum
AND
synack.seqNum + 1 = ack.ackNum
[...].
```

Once again, the query has to be changed to fit TelegraphCQ. Strangely, as mentioned in Chapter 4 the DSMS does not allow more than six conditions per query

Figure 5.6 The final query for the connection identification

```
SELECT
        syn.sourceIP, syn.destIP,
        syn.sourcePort, syn.destPort, wtime(*)
FROM
        streams.syn AS syn
        [RANGE BY '180 seconds' SLIDE BY '1 seconds'],
        streams.synack AS synack
        [RANGE BY '180 seconds' SLIDE BY '1 seconds'],
        streams.iptcp AS ack
        [RANGE BY '180 seconds' SLIDE BY '1 seconds']
WHERE
        syn.sourceIP = synack.destIP
        AND syn.destIP = synack.sourceIP
        AND syn.sourcePort = synack.destPort
        AND syn.destPort = synack.sourcePort
        AND ack.SYN = '0' AND ack.ACK = '1'
        AND synack.seqNum + 1 = ack.ackNum
GROUP BY
        syn.sourceIP, syn.destIP, syn.sourcePort, syn.destPort
```

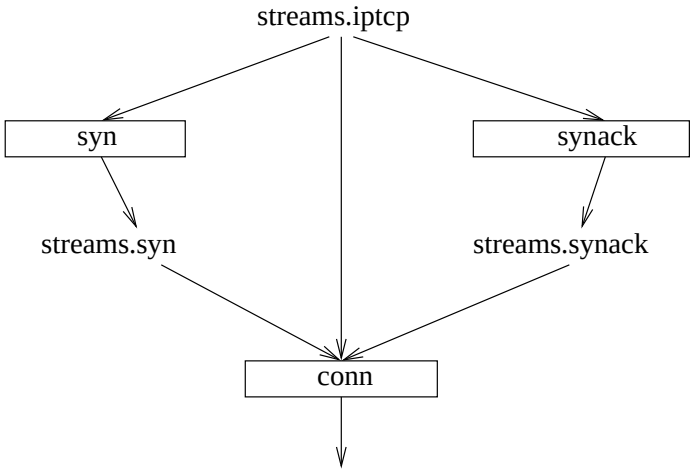
before the back end stops, and this limitation leads to yet another heuristic. We have to reduce the strictness of the `WHERE`-clause. The implementation is shown in Figure 5.6. As we can see, our heuristic based suggestion may cause an error in identifying a connection. The `ack` packet may come from another distinct connection in the data stream accidentally having an `ackNum` equal to `synack.seqNum+1`. Though, as such incidents do not occur regularly - having a 32 bits acknowledgment number - unexpected results may appear once in a while due to this heuristic.

Figure 5.7 gives an overview of how the queries work together to identify a connection. As we can see, all the three queries obtain tuples from `streams.iptcp`. The final query is called *conn*.

To ignore possible retransmissions that may appear in the real life Internet, the query also has to choose distinct results. Tests have shown that TelegraphCQ does not support the `DISTINCT` operator, forcing us to use the `GROUP BY` operator instead, which gives the same results [GMUW02]. An advantage of using that operator is the possibility of counting the number of retransmissions, if needed. This is done by including a `COUNT(*)` in the projection field.

The next challenge is to let the streams from these queries float in an efficient manner. Since TelegraphCQ has a limited sub-query support, we can try to find alternatives to the solutions we would have chosen in an ordinary DBMS supporting SQL99, by sending data back and forth of front end instances. However, since TelegraphCQ supports the `WITH` clause to some extent, we choose to use this feature

Figure 5.7 The overview of the connection identification. The queries are encapsulated in squares.



in all queries that need sub-querying. Thus, we have chosen to let the connection identification being located in one surrounding `WITH`-clause:

```
WITH
      streams.syn AS
      (
      <syn>
      )
      streams.synack AS
      (
      <synack>
      )
(
<conn>
);
```

The final query - conn - receives all the tuples from `streams.syn` and `streams.synack`.

Selecting the windowing technique is another challenge, since TelegraphCQ version 2.0 offers several. Though, in the following discussion we show that the windowing technique unfortunately leads to incorrect answers in the query.

As illustrated in Figure 5.5, the SYN and SYN/ACK queries are non-blocking and do not need to use windows to obtain the results. They simply select the packets that fit the conditions. On the other hand, the final connection identifier query is forced to use windows from the incoming streams to obtain the correct results.

To determine the ideal window size, which is a window identifying all three connection handshakes, we first need to know the TCP timeout interval. By TCP

timeout interval we mean the time it takes before TCP stops trying to send SYN packets to an unresponsive node during an active opening. In Linux, the default time is approximately three minutes. This corresponds to five retries. In Appendix B.4, we run a small test that confirms this. Linux has also a default of five retries for the *passive open*, i.e., the server's SYN/ACK reply to a SYN packet. Strict protocol implementation requires that we take these issues into consideration. Since our experiments mainly focus on performance evaluation, we choose to simplify the issue of connection initialization windows. We partly accept a timeout of three minutes, but this counts for the entire *open* operation. This implies that the *active* and the *passive* openings have 90 seconds each to get an SYN/ACK and ACK, respectively. As mentioned in Chapter 4, one second is the smallest update interval provided by TelegraphCQ, and we choose to let the window slide by one second. Generally, a connection manages to be established in much less than a second.

An issue that might be interesting to investigate is the maximum possible number of connections established every second, to estimate the memory usage for the connection identifiers. We use the assumptions from Section 5.2.3, which implies that a burst less 100 Mbits/s link transports 8,738.1333 packets of 1,500 bytes each second. When dividing the number of packets by three - as in the 3-way handshake - we get a maximum of 2,912.7111 possible connections per second[4]. The unique key needs at least 176 bits, since both the IP address variables use 56 bits each, and the ports use 32 bits, due to their `cidr` representation and TelegraphCQ's signed integers, respectively. Hence, the worst-case scenario requires approximately 512,637 bits, or 64,079 bytes, of new memory storage each second. A 60 seconds query window would require 3,754.6666 GB of storage, which is approximately 174,763 connections.

Fortunately, this enormous number of connections may not resemble the real life behavior of the Internet: When a connection is established, there is a high probability of succeeding data traffic between the two nodes. The scenario we described implies that all traffic is used to establish connections. Thus, all the connections will time out one minute after the connection has been established. Though, such scenarios do exist. As described in Chapter 2, some DoS attacks behave in a way similar to what we have described. Such attacks aim to block a server by sending SYN packets from different IP addresses and ports. The server sends the SYN/ACK packet and gets no response. It waits for a timeout period, and if it receives a substantial number of such packets, it stops serving other requests because of the overload. This issue is discussed further in Task 9 and Task 10.

Thus, conn uses sliding windows of three minutes when querying from `streams.syn`, `streams.synack`, and `streams.iptcp`. Ideally, since the query set identifies the open connections, we want conn to use a window size which guarantees that a connection is identified while it is up. This is a challenge, because small

[4]The theoretical maximum number of connections, using IPv4, is 2^{96}. 96 is the number of bits if we sum the IP addresses and the TCP ports in a packet.

HTTP connections may last up to a couple of seconds, while larger P2P sessions, video conferences, or other file transfers, like FTP, may last for several hours, depending on the network load and the file size. As we see in the following, this seems to be a rather challenging problem to solve.

As we know from Chapter 4, TelegraphCQ does not start to output tuples before it closes up to the time window. This means that all the connections established in the first three minutes of the data stream are not reported before after three minutes, i.e., the moment they slide out of the window. All other tuples live for maximum three minutes before they slide out.

There are several ad-hoc ways of solving this problem, but the solutions are either not correct, or not yet implemented in TelegraphCQ.

For the first ones, there is a possibility of sending connection tuples to a table instead of a stream. One solution could have been to set the stream to `ARCHIVED`. This would have stored all the tuples, but the stream can not be accessed using TelegraphCQ; one has to use the PostgreSQL interface, thus excluding this solution. Another possibility would have been to not use windows, thus storing tuples into the SteMs without ever deleting them. For a large number of connections, this can not be considered optimal, since connections are torn down, as well.

A better solution could have been to use the PSoup, as described in [CF03] and Chapter 4. One could have added connections to a data stem and deleted them when they were finished. Since PSoup is not included in our version of TelegraphCQ, we do not have the possibility of investigating this solution.

The next part shows how - given a correct modeling of the connection identifiers - the next set of queries simply use the conn information as a relation lookup, similar to Task 2.

As mentioned in the introduction of this task, a connection is identified by both the packets from the sender and the receiver. Therefore, we have to use two queries, one for each. This is because we can not use `OR` operators in the `WHERE`-clauses.

The first chore is to join these two streams with the conn stream. One of the streams' tuples has to be turned around, i.e., matched with the conn stream such that for example the conn's `sourceIP` equals the other stream's `destIP`. The streams' `totalLength`, `ipHeaderLength` and `tcpHeaderLength` are used as described earlier in this chapter. Figure 5.8 shows this turning, as tuples from `streams.iptcp` are joined with matching tuples from `streams.conn`. Query (a), which shows the client payload, does not turn any of the tuples. This is done by query (b). For example, the condition `conn.sourceIP = s.destIP` shows this.

The two resulting streams have to be merged together. This would have been solved by the `UNION` operator, but as shown in Appendix B.3, TelegraphCQ does not support this. Alternatively, we have chosen to simply join on the connection identifiers

and summing up the number of bytes. The final `GROUP BY` outputs each connection. All tuples within the ten seconds windows are calculated.

Figures G.1, G.2, G.3 shows the resulting query as used within the `WITH` clause. Figure 5.9 shows the packet float as it is described in this chapter.

The discussion above has given us at least three points that makes Task 3 theoretically hard to implement:

- The connections established during the three first minutes are not part of the connection results.
- The connections can not be stored in a table.
- For connections established after the three first minutes, they are only available for three minutes.

As the previous discussions have shown, we have chosen not to include the complete task in the performance evaluation. Still, we investigate the connection establishment, to see if it works correctly. This task shows that it is somewhat complicated to design queries in TelegraphCQ that reflect protocol states. This is further discussed in Tasks 5, 9, and 10.

5.3 Other Tasks

There are several other tasks to be discussed. These tasks do not work in TelegraphCQ as intended, or they can be considered equivalent to the first three tasks. These are described in the following, and can be used as a basis for future work. We discuss each of them and argue why the tasks are not used in the performance evaluation.

5.3.1 Task 4: How often are HTTP and FTP ports contacted?

Description

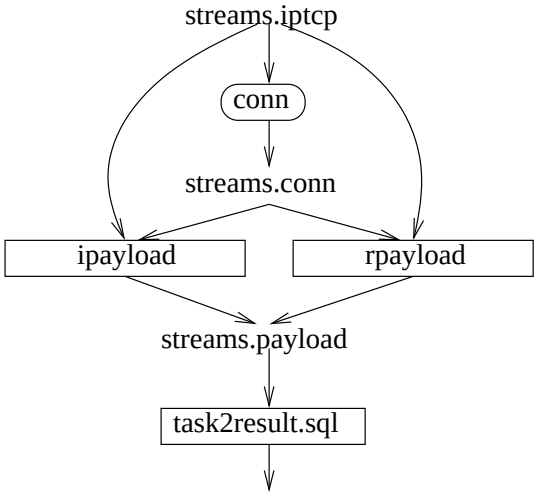
Protocols like HTTP and FTP are much used in the Internet, and are thus important to identify, so that the usage can be mapped more explicitly. We start by defining what the task is supposed to do.

By *often* we here mean the number of occurrences per tenth second. By *contacted* we mean when the TCP destination port matches the correct port(s) defined in the FTP protocol. The HTTP port is port 80, i.e., if a browser contacts a Web server, it automatically contacts port 80.

Figure 5.8 The sender and receiver queries that calculate the number of bytes.

```
-- (a)

(SELECT
        s.sourceIP, s.sourcePort, s.destIP, s.destPort,
        sum(s.totalLength) - (sum(s.ipHeaderLength)*4)
        - (sum(s.tcpHeaderLength)*4), wtime(*)
FROM
        streams.conn AS conn
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.iptcp AS s
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        conn.sourceIP = s.sourceIP
        AND conn.destIP = s.destIP
        AND conn.sourcePort = s.sourcePort
        AND conn.destPort = s.destPort
GROUP BY
        s.sourceIP, s.sourcePort, s.destIP, s.destPort)



-- (b)

(SELECT
        s.destIP, s.destPort, s.sourceIP, s.sourcePort,
        sum(s.totalLength) - (sum(s.ipHeaderLength)*4)
        - (sum(s.tcpHeaderLength)*4), wtime(*)
FROM
        streams.conn AS conn
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.iptcp AS s
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        conn.sourceIP = s.destIP
        AND conn.destIP = s.sourceIP
        AND conn.sourcePort = s.destPort
        AND conn.destPort = s.sourcePort
GROUP BY
        s.sourceIP, s.sourcePort, s.destIP, s.destPort)
```

Figure 5.9 The total overview of Task 3.



FTP is on the other hand more complex. Originally, FTP uses port 20 and 21, for data and commands, respectively. This has evolved, mostly because of the widespread use of the protocol, to involve more ports than 20 and 21 [PR85]. How the ports are managed and how FTP operates, is controlled by two different modes; *active* and *passive*.

The *active* mode lets the server use the original ports, while the client uses two consecutive unprivileged ports, i.e., port numbers higher than the well known ports as described in Task 2. The client sends a packet from the port n to the server's port 21. The server sends an answer to the client's port. Then the server sends a data packet to the client's port $n + 1$ from port 20, which is acknowledged by the client. The problem with this mode is that the server sends a packet not initiated by the client. This means that the packet most probably will be stopped by the firewall.

Though, this task has to be simplified. We know from Task 3 that the windowing may be a challenge also in this task. Thus we choose to re-define the problem to identify connections to port 80 and port 21 and focus on the problems concerning ways around the `OR` operator.

Query and Analysis

The HTTP connections are identified by selecting the packets having port 80 as destination ports. The FTP connections are identified by contacts to port 21. Thus the best solution would have been:

```
[...]
WHERE
        streams.iptcp.destPort = 80
```

```
OR
        streams.iptcp.destPort = 21
[...]
```

Since `OR` is not allowed in TelegraphCQ, we have to find a way to go around the problem. In Task 3 we chose to solve this by joining `streams.rpayload` and `streams.ipayload` on IP addresses and TCP ports. This is not possible in this case, since there are no similarities as in Task 3. A solution would have been to join on timestamps, and since we use punctuation tuples, this works even if there is no FTP traffic (using some pseudo code):

```
WITH
        streams.http AS
        (<select http packets>)
        streams.ftp AS
        (<select ftp packets>)
(SELECT
        http.dstPort, sum(http.number),
        ftp.dstPort, sum(http.number)
FROM
        streams.http AS http
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.ftp AS ftp
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        http.tcqtime = ftp.tcqtime
GROUP BY
        http.dstPort, ftp.dstPort)
```

The problem using this solution is that the timestamp's granularity is smaller than a second, so any chance of matching the joins is somewhat small.

Though, we have discovered another solution as well. We simply send both the HTTP and FTP packets to the same stream, `streams.task9`. This emulates a `UNION`. The final query does not need to do more than grouping on the destination ports. The query is shown in Figure 5.10.

A result from a browser session, where we browsed through HTTP pages and FTP directories is further described in Appendix H.1.

5.3.2 Task 5: For how long does a connection last?

Description

The discussion in Task 3 discloses some of the problems of storing the stream results. As stated, there are problems with removing these results if they are stored

Figure 5.10 The query that finds the number of times HTTP and FTP ports are contacted.

```
WITH
        streams.task4
        AS
        (
        SELECT
                destPort, count(*), wtime(*)
        FROM
                streams.iptcp
                [RANGE BY '10 seconds' SLIDE BY '1 second' ]
        WHERE                                                    10
                destPort = 21
        GROUP BY
                destPort
        )

        streams.task4
        AS
        (
        SELECT
                destPort, count(*), wtime(*)                     20
        FROM
                streams.iptcp
                [RANGE BY '10 seconds' SLIDE BY '1 second' ]
        WHERE
                destPort = 80
        GROUP BY
                destPort
        )

(SELECT                                                          30
        task4.dstPort, sum(task4.number) AS num, wtime(*)
FROM
        streams.task4 AS task4
        [RANGE BY '10 seconds' SLIDE BY '1 second']
GROUP BY
        task4.dstPort);
```

in a temporary relation as well. Closing down a connection is somewhat more complex than creating one, as we see below.

Postel [Pos81b] describes three possible close down situations, one for each of the two nodes in a connection, and one if both sides decide to close down simultaneously. We show the first two close-downs in the following. When the one of the nodes, *Node 1* initiates the close down, it sends a FIN packet to the other node, *Node 2*. At the same time, it enters a `FIN-WAIT-1` state. Node 2 receives the FIN packet and returns a FIN/ACK packet. Node 1 enters the `FIN-WAIT-2` state. Then Node 2 sends a FIN packet that is acknowledged by Node 1. When Node 2 receives the ACK, it sets the connection to a `CLOSED` state. Node 1 waits for 2MSL, i.e., per default 4 minutes; the expected lifetime of a packets sequence number before it is wrapped around. This implies that a new connection on the same set of addresses and ports is not established. When no packets have arrived from Node 2, it sets the connection to `CLOSED`.

When both nodes close down the connection simultaneously, they send the FIN packet and acknowledge those packets. Afterwards, they wait for 2MSL before they set the connections to `CLOSED`.

Query and Analysis

Implementing this behavior in a continuous query language resembles the connection identification. This means that we select all the packets having FIN set, and are part of the connection identification. We also select the ACKs on those packets. We select the similar behavior initiated by the other node as well.

This is initially written in the context of two queries using the `WITH`-clause, one for each of the two initiators. Thus, one of the queries switches the FIN packets and the other one switches the FIN/ACK packets. Finally, there is a query joining on the sequence and acknowledgment numbers. Unfortunately, TelegraphCQ closes down its postmaster if we try to run the queries. We do not investigate the reason for this. This task does not give any new information with regard to implementation; it uses much of the same building blocks as in Task 3. Thus, we choose not to investigate this task further.

5.3.3 Task 6: How many bytes are exchanged over the different connections during each week?

Description

This task is created to discuss the sliding window technique, which was the only alternative given in TelegraphCQ version 0.2. Plagemann et al. [PGB+04] concluded that such a task would be impossible to accomplish practically because the

memory usage would be considerable. The issue has to be considered with regard to the number of possible connections, where the network is monitored, and the packets' arrival rate. In Task 3, we show that the connection identification is hard to achieve due to the window semantics. Such a design excludes connections lasting over three minutes and favors shorter ones. Thus, the maximum length of results we see from one connection is three minutes.

It is also interesting to discuss whether or not the current task falls under the application domain of on-line network monitoring. Since the result is calculated once each week, the best solution would have been to store the relevant tuples on disk and perform a calculation once every seven days, even though the storage issue has to be taken into consideration.

The interpretation of letting the DSMS calculate all changes during a week, i.e., establish one or more aggregating windows that all together hold a week's information, is discussed in the following section. A straight forward implementation may require a vast amount of memory, as we will see, and the design resembles Task 3.

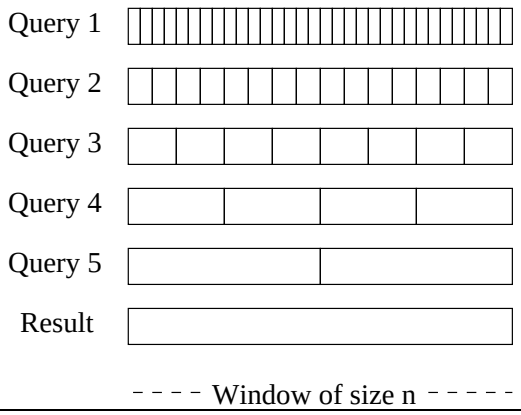
Query and Analysis

Conn, which was discussed in Task 3, is used to identify the connections in this task as well. Because of its inaccuracy - with regard to the three minutes lifetime of a connection - conn will still not be able to identify the longer ones. There also has to be a join between conn and the two directions, as designed in Task 3.

One advantage of calculating once every week, is that such a calculation can be assembled and summed up by several smaller ones. This can reduce the memory usage exceptionally. In Figure 5.11 we see how this can be designed; a binary division of the one week window. Already in the first set of windows all the packets that do not belong to any connections are filtered out. The resulting stream is sent to the next query. Since the first query performs some calculations in summing up the number of bytes, the next query only needs to sum these sums. A simple implementation is shown in Figure H.1. The even simpler implementation, using seconds instead of days, stop the postmaster. The log file complains that the enqueueing of the tuples did not work. By manually inserting tuples to the TelegraphCQ, the query worked, i.e., at least it did not stop. This probably means that the network load affected the execution.

It is important to remember that this task also relies on the ability of conn to give correct results. Some further testing of the task is described in Appendix H.2.

Figure 5.11 Binary division of the windows.



5.3.4 Task 7: What department has used how much network load on the university backbone in the last five minutes?

Description

This task is a derivation of Task 2. In [PGB+04], the discussion was centered around the "»" *cidr* operator. The operator looks into a given IP address and finds out if an input address is contained within [Pos]. For example, the expression `'192.168.1/24' » '192.168.1.5'`, states that `192.168.1.5` is part of the `192.168.1/24` subnet. Plagemann et al. [PGB+04] claimed that this operator gave wrong results. In Appendix B.1 we show that this is still an issue in TelegraphCQ version 2.0, probably because of the hash indexing in the SteMs. Thus, we are forced to use the same work-around as was introduced in the article; we use the "=" operator leading to large tables, and inaccurate answers. The inaccuracy is derived from the fact that now we have to store all the addresses in a table, and group by those addresses, just like for example Task 2. This does not give us information about the subnets; it gives us information about each destination address.

Query and Analysis

The query does not differ much from the one presented in [PGB+04]. The memory consumption will be almost the same as in Task 1 and the windows will still be sliding. Figure B.1 shows how the "»" operator is implemented in the task.

5.3.5 Task 8: For each flow, how many percent of the total load has been occupied during the last five minutes?

Description

Since we do not manage to identify the connections properly, we focus on the heuristic presented in [PGB+04] concerning the connections. The total load is to total sum of `totalLength` header fields for all packets during the five minute window. We simplify the query by investigating only the destination ports. This means that one port can be shared by many IP addresses without noticed by the query.

Query and Analysis

The total sum has to be calculated as a sub-query and then sent to the main query. We choose to sum up in a window:

```
WITH
        streams.totalLength
        AS
        (
        SELECT
                SUM(totalLength), wtime(*)
        FROM
                streams.iptcp
                [RANGE BY '300 seconds' SLIDE BY '1 second']
        )

        streams.partLength
        AS
        (
        SELECT
                destPort, SUM(iptcp.totalLength), wtime(*)
        FROM
                streams.iptcp
                [RANGE BY '300 seconds' SLIDE BY '1 second']
        GROUP BY
                destPort
        )
```

The main query ideally performs a join between the connections stream and total stream:

```
(SELECT
```

```
        destPort, ROUND((SUM(partLength.partLength)*100)
        /ROUND(SUM(totalLength.totalLength), 3), 3)
FROM
        streams.partLength AS partLength
        [RANGE BY '300 seconds' SLIDE BY '1 second'],
        streams.totalLength AS totalLength
        [RANGE BY '300 seconds' SLIDE BY '1 second']
WHERE
        partLength.tcqtime = totalLength.tcqtime
GROUP BY
        destPort);
```

As we see, by grouping by the destination port, we get a distribution of the different connections and how much percent they have used. The result is further referred to in Appendix H.3.

5.3.6 Task 9: How many connection initiatives have been rejected during the last ten seconds?

Description

This task investigates the number of SYN packets that have not gotten a SYN/ACK packet in return. This task is inspired by some of the queries posed in [JMSS05], which shows examples of network monitoring used in Gigascope [CJSS03].

This task and the following are posed as examples of how to use the DSMS in intrusion detection or to identify that some servers are down or not responding. This query investigates the last issue, i.e., comparing the number of SYN and SYN/ACK packets.

Query and Analysis

We aim to keep this query as simple as possible. Thus, we count the number of packets of SYN and SYN/ACK. If the difference is significant, something might be wrong in the connection establishment.

The two sub-queries are located within a `WITH`-clause (the new streams are defined with a prefix "2" to avoid confusions with Task 3):

```
streams.syn2 AS
(
SELECT
        'SYN', COUNT(ACK), wtime(*)
FROM
```

```
        streams.iptcp
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        SYN = '1' AND ACK = '0'
)

streams.synack2 AS
(
SELECT
        'SYNACK', COUNT(ACK), wtime(*)
FROM
        streams.iptcp
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        SYN = '1' AND ACK = '1'
)
```

The final query joins the two resulting streams on the timestamps they have created:

```
(SELECT
        count(syn.acks) - count(synack.acks), wtime(*)
FROM
        streams.syn2 AS syn
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.synack2 AS synack
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        syn.tcqtime = synack.tcqtime);
```

We expect that the difference between the two numbers is 0. This is also the case in the test runs. However, to investigate the accuracy, we *add* the two counts instead of subtracting them. We expect that the sum increases by two when much connection establishments are handled and decreases when no connections are established. Though, we experience that the sum increases to 20, and stays there, even when there are no more SYN and SYN/ACK packets in the network. The location of these result files is described in Appendix H.4.

Further investigation of this task may be interesting for future work, since the task is important with regard to identifying rejected connection requests.

5.3.7 Task 10: Identify TCP SYN packets for which a SYN/ACK was sent, but no ACK was received within a specified bound of two minutes on the TCP handshake completion latency

Description

This task aims to locate SYN flood DoS attacks, and we are interested in knowing which connections that have not received any confirming ACK packets within two minutes. This means we have to identify all SYN tuples leaving the window after two minutes, and see if there are any matching <SYN, SYN/ACK> intermediate tuples. If there are no such tuples, the leaving SYN tuple should be output.

In the STREAM DSMS [ABB+04], this can be solved by using two queries, one that obtains all the matching SYN and SYN/ACK packets, and one obtaining all the SYN/ACK and ACK packets. These two can be differed using the set operator `EXCEPT`, as done in [Her06]. In Appendix B.3 we show that this does not work in TelegraphCQ. Summing up, we need the following functionality:

- *All matches in a join are deleted from the stems, leaving only tuples that have not been joined.* This is an alternative to `EXCEPT`, and cannot be solved in the current version of TelegraphCQ. This is also an important feature with respect to the statement posed by Terry et al. [TGNO92] and referred to in Chapter 3: *We want to get all the messages that have* not *been replied to.*
- *All tuples that leave the window can be optionally displayed.* STREAM supports a DSTREAM definition that shows the deleted, i.e., leaving, tuples.

Since none of these two requirements are not supported by the current version of TelegraphCQ, there is no further foundation to further elaborate this task.

5.3.8 Task 11: Block all UDP traffic, and TCP traffic on port 6881

Description

This task wants the DSMS to play a role as a stateless firewall, which simply stops the traffic that is not allowed to pass. This is a new application for the DSMS with regard to the prior tasks. UDP packets are not connection oriented like TCP, and may result in network congestion and/or unwanted duplicates in a network. Thus, many network administrators do not want UDP packets in their network. Port 6881 is a much used BitTorrent port, implying that something illegal is downloaded. The last statement is wrong, of course; BitTorrent can be used to download all kinds of files, but we use it as an example.

`fyaf` can be configured to send both UDP and TCP packets to TelegraphCQ. For reasons of simplicity, we send the packets to `streams.iptcp`. We show why in the following.

We have chosen to implement the query using a pipeline. Firstly, we project all TCP packets, and secondly, all TCP packets that are not heading for port 6881. The query is implemented as follows:

```
WITH
        streams.task11
        AS
        (
        SELECT
                *
        FROM
                streams.iptcp
        WHERE
                protocol <> 17
        )

(SELECT
        *
FROM
        streams.task11
WHERE
        destPort <> 6881);
```

A simple test showed that the query gave the correct results. Note that even if TelegraphCQ does not support non-equalities in joins, simple selections are handled by the grouped filter, thus we get the correct results. Though, as the task is a simple projection, we do not test this task any further. The results are referred to in Appendix H.5.

5.3.9 Task 12: Which are the 10 most used destination ports, and how many packets have been sent to them during the last five minutes? Only a number of packets higher than 100 is interesting.

PostgreSQL uses the LIMIT statement to select the top values from a table:

```
SELECT column FROM table
LIMIT 10
```

Unfortunately, the eddy reports that it does not know how to handle a query plan containing the `LIMIT` statement, so we are forced to change the problem description.

We can e.g. use the `ORDER BY` statement to return all the destination ports ordered by the number of packets. If we use a single query without sub-queries, we have to perform a count on the the fly:

```
ORDER BY
        COUNT(*);
```

This makes the postmaster stop while testing. Thus, we have to use two queries to solve this task; one that counts and one that orders by the counts. We solve the requirement of only a number of 100 packets is interesting by writing

```
GROUP BY
        streams.iptcp.destPort
HAVING
        COUNT(streams.iptcp.destPort) > 100
```

The sub-query is implemented as following:

```
WITH
        streams.task12
        AS
        (
        SELECT
                streams.iptcp.destPort,
                SUM(streams.iptcp.destPort),
                wtime(*)
        FROM
                streams.iptcp
                [RANGE BY '10 seconds' SLIDE BY '1 second']
        GROUP BY
                streams.iptcp.destPort
        HAVING
                COUNT(streams.iptcp.destPort) > 100
        )
```

The main query then obtains the results from the sub-query:

```
(SELECT
        task12.destPort, task12.number, wtime(*)
FROM
        streams.task12 AS task12
        [RANGE BY '300 seconds' SLIDE BY '1 second']
GROUP BY
        task12.number, task12.destPort
ORDER BY
        task12.number);
```

The query is supposed to order by the number of packets from `streams.task12`. The problem is that when the packets arrive they are located within their own timestamp. This means that when `ORDER BY` is used, TelegraphCQ orders by all the tuples in the window, resulting in a correct order, but not grouped by the task number or destination port as intended. We choose not to use this task in the performance evaluation, because of those unexpected results.

The results from a run using the queries above are further referred to in Appendix H.6.

Chapter 6

System Implementation

In evaluating the performance of TelegraphCQ, we have to implement a system - an *experiment setup* - which runs the experiments in a deterministic and predictable way. It is also important that parameters, which may play a role in the total evaluation, are presented and discussed. As we see, such parameters may be the scripts that copy all results to the correct directories, as well as monitoring programs that monitors, e.g. CPU utilization.

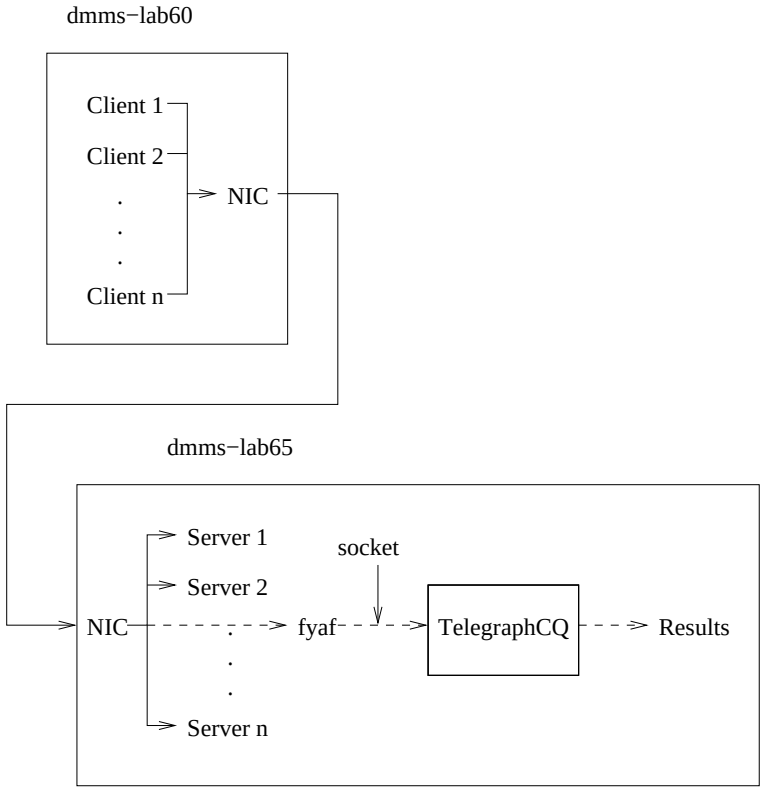
Thus, this chapter introduces the programs and parameters that make it possible to run the performance evaluation. We firstly define the system, i.e., the experiment setup, e.g. how we set up the computers. We also describe the packet filter `fyaf` and discuss how it is implemented and how it transforms the packets. Monitors and scripts used in the performance evaluation are described as well. Finally, we introduce other parameters which may play a role in the total evaluation.

6.1 Definition of the Experiment Setup

By *experiment setup*, we mean the operating system, monitoring tools, filters, and calculation scripts. In other words; the experiment setup system is everything except from TelegraphCQ and its processes. From this point, we use the terms *system* and *experiment setup* interchangeably to describe this.

The goal is to have complete control of the setup and its performance during the experiments. Thus, connecting a computer to the network and analyzing the real life streams may not be sufficient. Firstly, these experiments are not reproducible; the data streams are never the same from a day to another. Neither do we control the users' and systems' behavior on the network. Secondly, the characteristics of the current network may not be representative for networks on the whole, and thirdly, the network may not have the characteristics we are interested in when analyzing different parts of the DSMS at the different tasks.

Figure 6.1 The experiment setup for the performance evaluation.



To further increase the control over the experiments, the experiment setup only consists of two computers; `dmms-lab65` and `dmms-lab60`. They are disconnected from the network and form their own small network by having a network cable stretched between them. Each computer has two Intel Pentium 4 processors, each yielding 3 GHz. Each of the two computers also have 1 GB memory. Each machine runs with Linux SuSE 9.2 with a 2.6 kernel.

The experiment setup can be seen in Figure 6.1. `dmms-lab60` has the ability of sending data from up to n different clients. The data is received at up to n servers on `dmms-lab65`. To generate traffic, we use a public domain traffic generator, which is introduced in Chapter 7. A filter, `fyaf` - which has been introduced in the preceding chapters and is thoroughly described and discussed in Section 6.2 - obtains the packets from the NIC, and sends them to TelegraphCQ in a CSV format. The results from TelegraphCQ is then stored to files.

What we have mentioned so far implies that we have chosen to let the DSMS listen to all traffic from and to an end node, e.g. a server. The alternative design would have been to place the DSMS on a computer acting as a bridge or at a router. We obtain all the packets that are sent and received at the NIC. Thus, in our experiments, the DSMS would not have received any more traffic than if it was

located on a bridge. The only disadvantage is that the DSMS has to share CPU time with the servers. Fortunately, as shown in Chapter 7, the servers only act as network sinks, i.e., they do not perform any operations except from accepting and listening to the sockets. Though, they write packet information to file as well. The possible overhead is further discussed in the performance evaluation.

6.2 Filters, Monitors, and Scripts

A monitor is used to investigate the program that is analyzed. Monitors are also used to measure the experiment setup, like CPU utilization, for example. Jain [Jai91] lists several types of monitors, but since our study is based on black-box measurements, the monitors discussed in this section aim to investigate the experiment setup. TelegraphCQ monitors itself as well, but such monitoring may not be correct. Hence, we investigate one of the monitors in Chapter 7.

In our experiments, we need to monitor the workload. The evaluation is based on sending data packets at differing rates to analyze the performance of TelegraphCQ,

As mentioned above, we use scripts to run the experiments. The intention is to make these experiments as reusable and correct as possible[1]. Since we use data streams that last for a given period of time and then stop, we need the system monitors to cope with this event, stop monitoring, and report the results. Concretely, we need `fyaf` to understand that the stream is finished and that results can be reported. We also need a script that stops the remaining system monitors when necessary.

We have several alternative solutions. As in [PGB+04], we let the DSMSs monitor themselves by reporting the number of tuples they have processed. But we also need the monitors measure the amount of packets the DSMSs receive and send. We would also prefer that the monitors to measure the time the DSMSs take; from the first tuple arrives to the last tuple is sent.

6.2.1 `fyaf`

Krishnamurthy et al. [KCC+03] states that the future releases of TelegraphCQ will contain a functionality for specifying the data stream filter. Potter's wheel [RH01], another project at Berkeley, is a GUI based program developed for DBMSs to transform data items to fit the CSV format accepted by the DBMS. In network monitoring, there is need for a tool that filters raw data packets so that it e.g. fits the stream as defined in Chapter 5. This has to be performed in real-time.

As mentioned several times throughout this thesis, `fyaf` is a simple filtering and monitoring tool we have developed to fit these needs by mainly focusing on transforming raw data packets delivered from the packet capturing interface, `pcap`

[1] Thus, we cannot, and will not risk the inaccuracy of using a stopwatch and press `Ctrl-c` after t minutes of experimenting.

[pca], to the CSV format accepted by the DSMSs. In addition to filtering, `fyaf` also monitors the stream and reports the results when finished. This section shows how `fyaf` is designed and how the packets are transformed according to the corresponding packet headers.

`pcap` dumps packets directly from the NIC, thus, the packets are not significantly affected by the operating system's protocol stack, as they would have been if we communicated through sockets. This means that `pcap` gets raw and unprocessed binary data packets. The packets look like the ones in the top left box in Figure 6.2. The only difference is that they arrive as a byte array without the *newlines* at every 32 bits. We have, for simpler reading, added the newlines in the figure. `fyaf` starts a thread that reads new packets from the packet capturer.

When the packet is received, the header is assumed to map an *Ethernet* packet. We use this assumption to identify the first header. As shown in Appendix J, all the other header types are mostly identified in the prior header. The pointer to the first header is therefore sent to an Ethernet packet `struct`. This struct is mapped directly to the pointer, and by doing that, the header fields are accessible by using the `struct`'s variables.

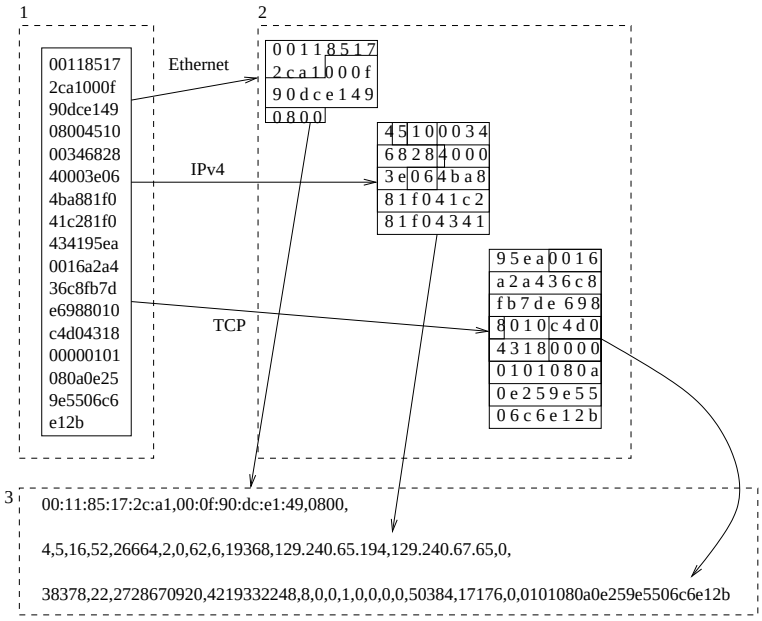
Section I.1.1 and Section I.1.2 in Appendix I show the header and source file of the definition and printing/mapping of the IP header, respectively. In `struct IP4`, we see that the lowest granularity used is the 16 bits unsigned short. All header fields smaller than 16 bits are mapped together. This is done mostly because of the big endian representation of the data when it comes from the NIC. We originally wrote `u_char version:4;`, which gives four bits. This functionality is allowed, but not ANSI C compliant. Since C only provides the functions `ntohs` (Network TO Host Short, 16 bits) and `ntohl` (Network TO Host Long, 32 bits), we are forced to use 16 bits as the smallest representation. In the top right box in Figure 6.2, we have illustrated the mapping. We see that the different headers are projected into the packets. The supported header types implemented in `fyaf` is Ethernet, IPv4, TCP, and UDP. Except from the UDP header, all the other headers are used in the experiments. Ethernet is used because of the network we are using. IP and TCP are used since we define both headers in `streams.iptcp`.

When the mapping is finished, the `struct` types are written to a buffer. This buffer is filled with the headers the user wants to analyze. When the header mapping is finished, the buffer is sent to `stdout` and/or a socket, depending on the user's preferences. The `fyaf` source code is available in the DVD-ROM.

`fyaf` monitors itself by running two separate threads, Thread 0 and Thread 1. Since `fyaf` only runs during the experiments, it needs to find out when the experiment is finished and to shut down cleanly.

The monitoring is activated by the arrival of a packet. For each of the input packets, a timer is updated. When the input is registered, Thread 1 waits for new input. Meanwhile, the main thread, Thread 0, measures the time, sleeps for a specified time-to-live (TTL) interval, and checks if Thread 1 has performed any registration

Figure 6.2 The packet mapping and transformation in `fyaf`.



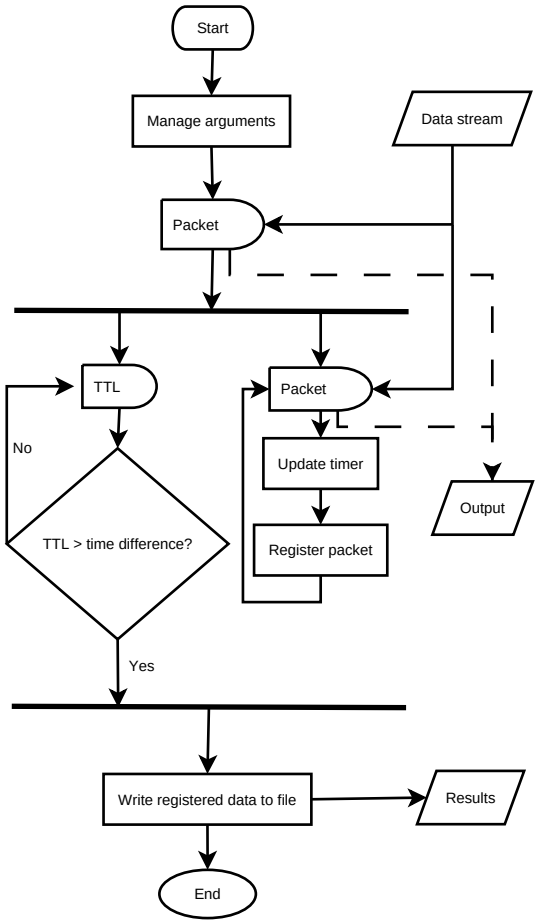
while Thread 0 was asleep. If not, Thread 0 calls a function that reports the results and terminates the program. This implies that the monitor does not spend any time for unnecessary I/O during the very run time of the experiment. The TTL is defined by the user. Figure 6.3 gives an overview of the flow. We see the parallel processing of the two threads between the two thicker horizontal lines.

The monitors in `fyaf` are considered as *batch monitors* [Jai91], i.e., they do not report anything before the monitoring has ended. As mentioned, we have chosen this solution because we want to reduce the I/O overhead real-time reporting may add. We are not interested in analyzing the data in real-time. Since this is a performance evaluation, it implies that we analyze the data after the experiments are finished running and the scripts have calculated the results.

As an alternative, we could have used packet capturing tools like `tcpdump` to obtain the packets, pipe them to a set of Linux command-line scripts, like `sed` and `awk`. The result could have been sent to `source.pl`, which is mentioned in Chapter 4. `source.pl` sends the tuples to the TelegraphCQ wrapper. However, as we start using pipes, it is harder to specify whether the scripts provide bottlenecks in the system. Neither, do we get any statistics about e.g. the number of packets sent through the pipes. Thus, to keep the system simple and lucid, `fyaf` is used.

Since we focus on reducing the system overhead, we have written `fyaf` in the C programming language. This reduces the risk of any unknown overhead that may follow other higher level programming languages.

Figure 6.3 The structure of the monitoring threads.



6.2.2 System Monitors

Aside from `fyaf`'s monitoring of the traffic, we have also added three system monitors that report on CPU and memory utilization, for example.

- `vmstat` reports general statistics of the virtual memory, I/O, and CPUs in the system. The statistics show for example available memory each second. This feature makes it simple to generate plots; one can read directly from the dump file without much parsing.
- `top` reports among other things how much memory and CPU each process consumes per second. Investigating one single process is done by using e.g. the program `grep` to filter all lines having the process's PID.
- `sar` reports almost everything in the system. The strength in `sar` is that, besides from monitoring the memory, like `vmstat`, it also monitors e.g. paging, interrupts, and socket activity. Since we write to file, process information, like what `top` returns, is not stored, according to `sar`'s specifications. `sar` aims to write a considerable amount of data to file for each second. Unfortunately, it does not seem like `sar` is consistent in these matters. We observe that the timestamp written by `sar` not always reports each second. Sometimes the timestamp skips two seconds. This behavior has to be considered when we use `sar`'s results in the discussion of the results.

There is some overlap in the monitor results. `sar` reports everything reported by `vmstat`, but in the design and implementation period, we added `sar` in a later stage than `mvstat`, so we kept `vmstat` because of its backward compatibility and simple reading. Fortunately, as the information gathered from `top` is not stored in `sar` when the output to file option is used, the overlap is minimal in that case. Further description of the monitoring tools can be achieved from the Linux `man` pages and Johnson et al. [JHP05].

As we further discuss in Section 6.3, there has been a slight overlap of running instances of the system monitoring tool at the beginning of each experiment. We can only monitor what happens during the run, because of this issue. The scripts turn off TelegraphCQ only about a minute after the streams are finished, reducing the ability to investigate possible emptying of the DSMS when it was supposed to be finished.

6.2.3 Scripts

Following is a description of the experiment scripts we have implemented to help us perform the experiments correctly. They play an important role in the experiment setup, since none of the experiments are done by hand. The scripts are located on both machines, and each has responsibility for different part of the experimental process.

- `dmms-lab65` scripts:
  - `sscript.pl` contains all the tasks that are run, the number of connections, and network loads. This is the only script the users interact with when defining the factors for the various tasks.
  - `superscript2.pl` is called by `sscript.pl` and starts and stops the DSMS.
  - `experiment_client.pl` is called by `super_script_2.pl` and starts `fyaf`, system monitors, and connects to `dmms-lab60` to start the scripts on that machine remotely.
  - `create_servers.pl` is called by `experiment_client.pl` and defines the running time of the servers.
  - `tg_server_run.pl` is started by `experiment_client.pl` and starts the server waiting for data packets from `dmms-lab60`.
- `dmms-lab60` scripts:
  - `experiment_server.pl` is called by `experiment_client.-pl` to establish a connection between the two machines. It also receives startup commands from `experiment_client.pl` to start the following scripts.
  - `change_template.pl` calculates the variables sent by `dmms-lab-65` that are used to setup the data traffic.
  - `create_clients.sh` is called by `change_template.pl` to change the setup files with respect to for example protocol, header size, and network load.
  - `tg_clients_run.pl` is called by `experiment_client.pl` to start sending data.

When `super_script_2.pl` starts TelegraphCQ, it calls a small start-up script called `run.pl`. This script sends the current task's query into a front end process and writes the results to a file.

The idea behind having more scripts with defined responsibilities and input parameters, is that we can easily replace single scripts with updates. All the scripts are located in the DVD-ROM. See Appendix K for more details.

6.2.4 TelegraphCQ Monitors

We have chosen to include some of TelegraphCQ's self monitoring functionalities, i.e., the reporting of shedded tuples, and the introspective queries, in our performance evaluation.

To identify how many tuples have been shedded, we have used two queries, one obtaining the kept tuples, and another obtaining the dropped tuples. We use three introspective queries to obtain tuples from all three streams described in Chapter 4 and in Appendix C.1.

To solve this, we have added calls for the startup of the queries in `run.pl`. This assures that the monitoring queries are started each time, and that we do not need to start them explicitly.

All the monitoring queries are simple, in which they project all the attributes from all tuples in the streams.

6.3 Other Parameters

As shown above, we run several programs to filter and monitor both the system and TelegraphCQ. These programs affect the performance of the system and TelegraphCQ, but they also play an important role in the overall evaluation. Following, is a discussion of some of the parameters that may affect the evaluation and that we have not added explicitly.

As we can see, the experiment setup in Figure 6.1 may contain up to n servers. We have experienced that using more than one connection causes the traffic generator to add much to the CPU utilization. Thus, in these experiments we only use one connection for the on-line analysis. We consider that one connection is sufficient for Task 1 and Task 2. Task 3 needs more connections to give an impression of its applicability. Therefore, we have used the traffic generator to create a ten-connections stream to test the results. Hence, the analysis of Task 3 is tested mostly to verify the statements made in Chapter 5.

In the tests, we eliminate all other traffic except what is generated by the workload generator by only using a cable between the two machines. The only additional traffic we may observe are control packets like e.g. ARP packets [Plu82] that find the machines' MAC addresses. This may slightly affect the output from `fyaf`, but we have only experienced a minimum amount of packets. Since we are disconnected from the network, we also disable the firewall on the ports that we use in the evaluation.

User interruption is also reduced by using the scripts to run all the experiments. We believe that using scripts only cause minor overhead when the scripts wait for the DSMS to finish. All other calculations, e.g. calculating the average and creating graphs, are done between, or after the experiments are finished, as batch jobs.

Finally, as seen in Section 6.2, we also use system monitors that may add some overhead to the system. Though, we have experienced that the operating system, since the monitors have nice values approximately to zero, schedules the monitors such that they monitor longer than the experiment lasts. This means that we may experience double monitoring, i.e., two instances of the same monitoring

program runs concurrently for a five minutes time period. We have located the bug in `experiment_client.pl`. To make the scripts wait for each other, we use the `waitpid($pid, 0)` function to wait for `fyaf` and TG to finish. We also use this function to wait for the system monitoring programs. Unfortunately, the monitoring programs are forked within an already forked process leading the `waitpid($pid, 0)` function to return immediately. The intention was to let the monitors monitor the system for five minutes after each experiment was finished. The result is now that TelegraphCQ is shut down only after a short timeout interval and that we can not investigate the results. We assume that the overhead a double instance of `vmstat`, `top`, and `sar` may add to the performance of the system is minimal. As we see in the results of the performance evaluation, there are no signs of overhead in the current time period, and investigation of the `top` result file shows that the monitors mostly use between 0.0 to 0.1 in memory and 0.0 to 1.0 in CPU. We also tested a set of runs omitting the monitoring tools and experienced no differences in the results. The only major drawback is that TelegraphCQ is shut down too early to see for how long it continues dumping results when it is finished. Though, an indication may be given by investigating the `vmstat` output and possibly estimate its gradient for the approximately one minute it takes to wait for `fyaf` to time out and copy monitor result files to the result directories.

Chapter 7

Performance Evaluation of TelegraphCQ

This chapter describes and discussed the performance evaluation of TelegraphCQ. As described earlier in this thesis, the idea is to send data packets to TelegraphCQ and measure the system when it is run, and investigate the results, as well as the monitor output which is described in Chapter 6.

Firstly, we introduce the metrics and factors we use in the performance evaluation. The metrics aim to investigate the performance of the different requirements presented in Section 2.3. The factors are used to affect TelegraphCQ's response to the metrics. We also discuss the evaluation technique we use, and show how the traffic generator work.

Secondly, we show the experiments and the results, after investigating the reliability of TelegraphCQ's Data Triage tuple reporting and the overhead added by the introspective queries. We go through each task sequentially and show both the setup and the result. Each section in this chapter describes its purpose in the performance evaluation as a whole.

Finally, we conclude the performance evaluation based on the results.

7.1 Metrics

Jain [Jai91] defines a metric as a quality to which - in this evaluation - the DSMS and system is measured against. In our case, we have chosen two main metrics; *relative throughput* and *accuracy*. A third metric, *consumption*, is also evaluated. We describe each of the three metrics in the following.

- **Relative throughput.** *How much data does the DSMS manage to compute?* The relative throughput is the relation between the data rate TelegraphCQ

receives and the data rate it manages to compute. Since TelegraphCQ has a load shedding mechanism, it reports about the dropped and kept tuples.

Reiss et al. [RH06] integrate these results into their queries by selecting this information from the streams. In our tasks, we choose to obtain these tuples in separate queries and analyze the result afterwards.

- **Accuracy.** We look at each of the tasks and investigate the accuracy, i.e., do the queries give the expected results? We base the evaluation on the queries in Chapter 5. Investigating the accuracy for data streams is not trivial, as stated in Chapter 6; there are several parameters that affect the total result. This has to be taken into consideration as we investigate the accuracy in our tasks. Thus, as shown in Chapter 5 we investigate the accuracy of each task. At some of the tasks, we investigate the relation between accuracy and relative throughput as well.

- **Consumption.** How much overhead and resources does the DSMS require? We have chosen to split this part into two different consumption types.

  – **Memory.** We use `vmstat` to provide us with information regarding the memory usage. We compare e.g. relative throughput and memory usage to see if we observe any patterns.

  – **CPU.** The combination of CPU and memory consumption may be an interesting metric. The CPU utilization is investigated using `top`.

7.2 Factors

The factors are what we vary during the experiments. Since the performance evaluation is supposed to investigate what is happening when TelegraphCQ is under stress, we choose to focus on varying the *network load* and *number of simultaneous queries*. We have a slight main focus on the network load. We discuss each of these factors.

- **Network Load.** We define network load as the number of bits received at the NIC. Mainly, this factor will be increased from almost zero to 10 Mbits/s[1], implying that if `fyaf` manages to capture more data than the NIC, the NIC is the bottleneck. We also assume that `fyaf` performs faster than the DSMSs since it is less complex. This is verified by `fyaf`'s log files and the performance evaluation.

- **Number of simultaneous queries.** Sometimes, there is a need for more than one query running at the same time. An example is to run several tasks concurrently. This is also tested by running the queries that obtain the shed-

[1] We use the terms "Mbits/s", "Mbs", and "Mb/s" interchangeably.

ded tuples stream as well as the set of TelegraphCQ monitors described in Chapter 6.

The system has a possibility of sending data over several connections, and we originally intended to use the number of connection as a factor, but due to the running time of the experiments, we focus on the two factors mentioned above.

7.3 TelegraphCQ Configuration Files

We have slightly changed the `postgresql.conf` configuration file and `telegraphcqinit.h` to e.g. reduce the size of the CACQ bitmap and increase the queue sizes and number of queues. The size of the CACQ bitmap is reduced to match the maximum number of queries our tasks use. The queue sizes and number of possible queues are increased to make TelegraphCQ manage more tuples internally. For possible later re-tests to verify our results on other machines, the files are located on the DVD-ROM.

7.4 Evaluation Technique

For the performance evaluation, we use measurements, which means that we use the monitors and result files, and analyze these when the experiments are finished. Based on the complexity involved, e.g., the TCP layer in the operating system, and that we perform the experiments as black-box, evaluation techniques like analytical modeling might be difficult to achieve.

The results are shown as averages over a given number of similar tests. The average a is calculated by the following equation:

$$a_n = \sum_{i=1}^{n} \frac{ia_{i-1} + a_i}{i+1}.$$

This equation helps us calculate the average on the fly when reading through the result files. The results are shown in both tables and graphs. In situations where we use data from a whole run, we only show the graphs since putting all the data in tables requires a lot of space. As we see in some runs, the variability might be high, i.e., the results may vary significantly between each run, but in other runs we see a jagged curve even when the average is calculated. On some results we show variability by range, in trying to explain results that differ significantly from expected results.

For some experiments, the amount of data exceeds 1 Gbytes, which has made it necessary to implement several scripts to gather the correct information. These scripts calculate the average and generate graphs and tables. The scripts are located on the DVD-ROM.

7.5 Workload Selection

The workload we use for this performance evaluation is TCP packets sent at different network loads, i.e., Mbits/s. Since the experiments are based on network monitoring, network data is considered to be a satisfying workload.

The data packet traffic is generated by the public domain traffic generator called TG, which is developed at SRI International [MLD02]. TG provides functionality for sending a stream of packets consisting of pre-defined protocols with possibility of choosing between varying packet lengths and rates. As indicated earlier, the generator only sends one stream per client/server connection. Thus, we have to start more than one instance of the client/server couple if several connections are needed. This is done to create ten-connections data traffic, which is used in Task 3. We have opened 40 ports in the firewall on `dmms-lab65` and `dmms-lab60` so that we have 20 ports for TCP and 20 ports for UDP.

We have solved the changing of TG settings by creating a template file having general fields that are substituted by the correct values using the command line editing program `sed`. By using `sed`, we integrate the editing in a script. The template is located in the DVD-ROM for further investigating.

Note that TG's network load is determined by an inter-arrival rate of number of packets in seconds. The network load is therefore calculated as the following code example shows:

```
result = (network_load/packet_size_bit)/number_of_nodes;

inter_arrival_rate = 1/result;
```

We *add* bytes for Ethernet, IP and TCP to the packet size before calculating `result`. Since we define TCP's segment size, TG implicitly adds these header fields. We have also experienced that TCP's option fields are used, so we add 12 bytes to the 20 bytes TCP packets.

At high loads, the inter-arrival rate may be so small that the computers handling of floating numbers may affect TG sending rate. When having a network load of 1 Mbit/s (1,048,576 bits), 576+66 bytes (5,136 bits) packets and one single node, we see that the result is $1/\frac{1{,}048{,}576}{5{,}136}$ packets each second. The inter-arrival rate is therefore approximately 0.0049 seconds. If, for example, TG uses the `usleep()` function to wait between each packet, we see that the overhead might be on the expense of the accuracy. We investigate TG's accuracy in the following evaluation.

7.6 The Experiments

7.6.1 Design

In this section, we investigate and discuss the parameters and factor values we use. As of the parameters, we have packet size and rate characteristics, running time of one experiment, number of runs per experiments, number of connections, and protocol.

We instruct TG to send packets with 576 bytes at a constant rate. The packet size plays an important role in the performance evaluation, since we only measure the headers. This means that the interval between each header - the payload - must be of a size that may reflect the real world, but still stress TelegraphCQ. We also want to reduce the fragmentation in the TCP layer, since the system in the early stages sent a specified number of packets instead of sending for a time period. Postel [Pos83] sets the maximum segment size (MSS) for TCP to 576 bytes including the IP and TCP headers without options. However, not knowing that TG added the IP and TCP header to the specified size, we ended up having packets of 628 bytes instead. This probably makes the TCP layer fragment and defragment some packets. Though, we expect that this does not play a significant role in the evaluation in its entirety. Since we use TCP, the network traffic is also filled with ACK packets, and `fyaf` filters these as well.

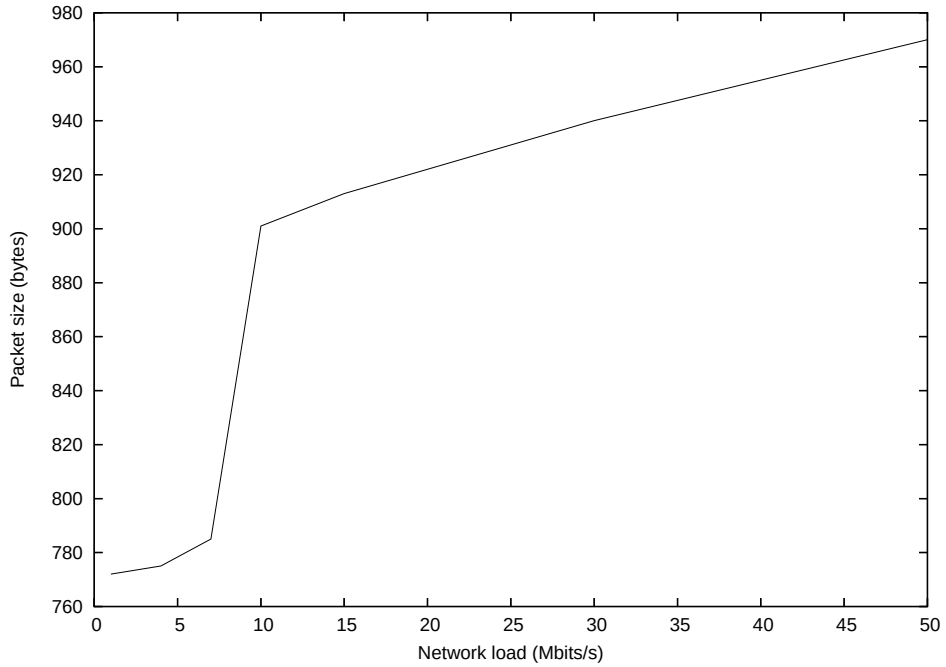
We have observed that the actual average packet size on the network is *not* 628 bytes, because of the fragmentation, defragmentation, and ACK packets. Figure 7.1 shows an example of the packet streams as they appear in our experiments. At 1 Mbits/s, the average packet size is 772 bytes, and increase to 785 at 7 Mbits/s. A sudden increase to 901 bytes at 10 Mbits/s suggests that the network load affects the average packet size. Probably, the TCP layer starts defragment packets at higher rates to reduce the total number of packets.

Note that `fyaf` is implemented such that only TCP packets are filtered. This means that other packets, like ARP are not filtered. Though, the number of control packets is so small, it is considered a negligible overhead in the network traffic [Her06]. An investigation of the packet header size is an issue for future work.

As TG is instructed to send at constant rates, we construct an environment that is presumably a worst-case scenario for the adaptive eddy mechanism provided by TelegraphCQ. As the eddy's strength is the per-tuple evaluation, a constant rate probably makes the eddy's adaptivity redundant and may also result in a lower performance in TelegraphCQ.

An experiment *run* is set to last for 15 minutes. Thus, the TG client sends packets for 900 seconds, but the server is open for 930 seconds. We instruct the scripts to let the server run longer than the client, to avoid that the server closes down too early. As seen in Chapter 5, the queries in Task 2 use windows that last for five minutes. To fully see a development, we are forced to run the experiments for that

Figure 7.1 The average packet size in test streams at varying network loads.



long. Fortunately, since the experiments are automated, these runs of 15 minutes are performed sequentially almost 24 hours a day, except for a couple of hours copying the results to other machines and performing backups. Each experiment is run *five times*, which leads to a running time of approximately 30 hours for a task having five different network loads. Even though we evaluate a small number of tasks, we have run several *sub-tasks* that are variations of the original tasks. This is clearly shown in the description of each task in the following experiments.

Neither the system nor TelegraphCQ are error free. Given these possible error runs, and that this thesis does not focus on debugging TelegraphCQ, we only use the results from perfect runs in the evaluations. This may affect the reliability of results where four out of five runs had errors. We sum up the error information in two tables for each task. One for TelegraphCQ's stopping and one for `fyaf`'s stopping without reporting any monitoring results. `fyaf` returns with error number 13, which indicates a `Permission denied`. This may be because `fyaf` is run as `sudo` because it uses `pcap` in promiscuous mode.

To reduce the number of factors, we limit the number of online connections to one. We also only use TCP packets in the experiments, even though the system also supports UDP packets. Creating UDP streams in TelegraphCQ is considered trivial since the header fields resemble TCP's header, only that there are fewer header fields, as seen in Appendix J. The disadvantage of TCP is that ACK packets affect

the total amount of the network load. This could have been solved by using UDP, but since the number of header fields are so small compared to the TCP header, we assume that using UDP affects TelegraphCQ such that it may perform better. We have not investigated this in our evaluation, and do not consider this an issue for further investigation in this thesis.

For the load shedding mechanism, we have chosen to use the `keep count` argument for the Data Triage tuples. This argument makes it easy to calculate the relative throughput after an experiment is finished. We use two queries that project all the information from the kept and the dropped tuple streams. As noted in [Loa], a slight overhead is considered in keeping a count for all the kept tuples as well, but we have to use these results to calculate the relative throughput. We investigate the possible overhead in the following sections.

Any possible parameter change is also noted and described in the respective tasks. In this thesis, this is mainly considered in the pre-task described below.

The following sections describe the experiments that are performed. We start by performing three preliminary tasks. The first preliminary task investigates the accuracy of the network load generated by TG, and how much of this load `fyaf` manages to handle before it starts dropping packets. The second preliminary task investigates the overhead of the system monitors with regard to CPU and memory consumption. The third preliminary task aims to investigate the possible overhead in using the introspective queries in TelegraphCQ. It also investigates the shedding mechanism.

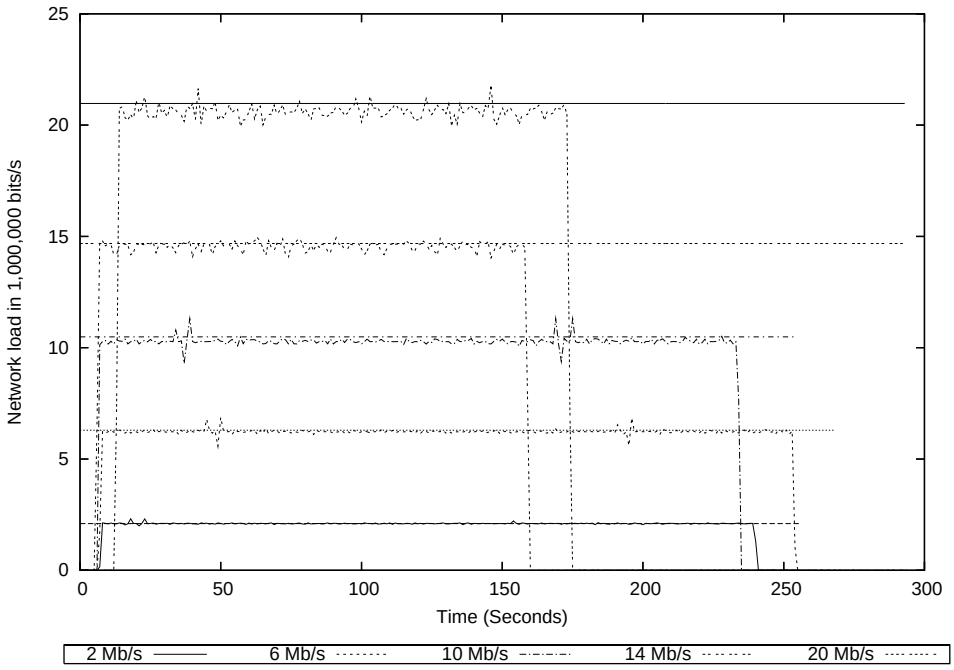
The next experiments correspond to the respective tasks. We show the setup and results for each task sequentially, since each task can be considered independent of the others. We end this chapter by summing up the results from the tasks.

7.6.2 Preliminary Task 1: Network load accuracy and `fyaf`'s relative throughput

Network Load Accuracy

Since much of the performance evaluation is based on an assumption that the network load reported is correct, we investigate this by looking at the result files from `sar`. One of the options in `sar` is to measure the number of packets and bytes that have entered and left the NIC. These results give a satisfying indication of the network load. As mentioned in Chapter 6, `fyaf` reads data packets directly from the NIC, thus, we consider the results from `sar` to be sufficient. We use the results from the third run in the `fyaf` tests described below. We investigate the throughput at 2, 6, 10, 14, and 20 Mbits/s. We have inserted the expected values as well, to compare. Figure 7.2 shows the result from TG. The y-axis represent the network load in 10^6. We see that at 1 Mbits/s (which is 1,048,576 bits), TG matches the expected value adequately. This is also the case at 6 Mbits/s. At 10 Mbits/s, the data

Figure 7.2 Network Load accuracy showing the reported and the expected network load.

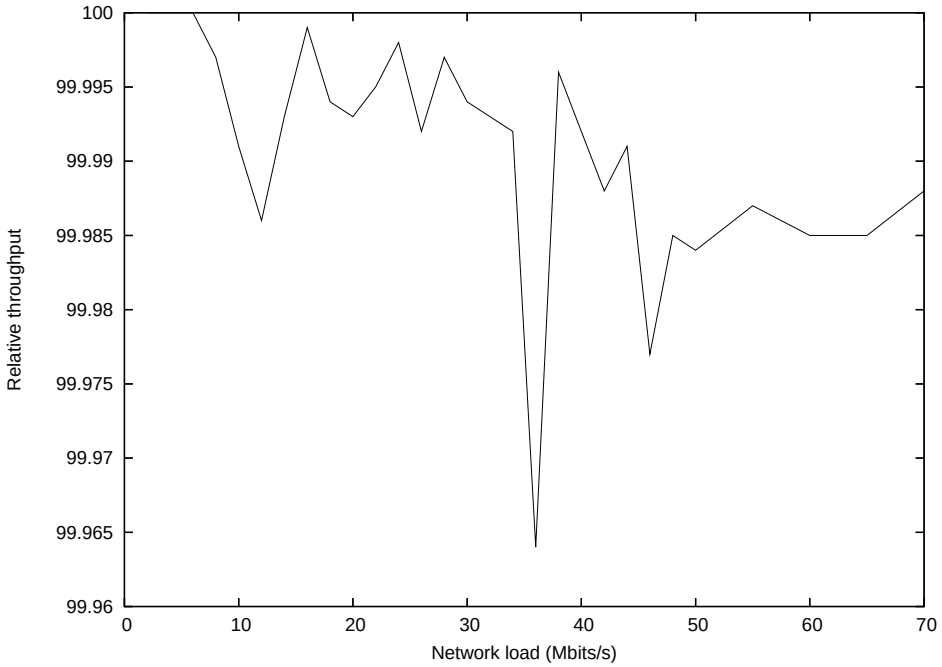


starts arriving more bursty, and has a lower rate than expected. This trend seems to continue when the network load increases. We conclude that TG sends network data in an accurate rate up to 10 Mbits/s.

`fyaf`'s Relative Throughput

We have inserted `fyaf` into the system such that network data is filtered correctly. It is important that `fyaf` manages to process the arriving packets such that it is *not* a bottleneck in the system. In Chapter 6 we imply that `fyaf`'s simplicity makes it easy to see whether the packet filtering is a bottleneck in the system. If it is a bottleneck, we need to identify this as a possible source of error. Hence, we run a set of five minutes tests where `fyaf` packets generated by TG, and sends them to a TCP sink instead of the DSMS. TG sends at different network loads. Figure 7.3 shows that the relative throughput is high. The average relative throughput seems to drop as the network load increases, but it does not keep below 99.975% in the investigated load range. We only observe an outlier at 36 Mbits/s, but we do not investigate this further. We conclude that within the range of 1 Mbits/s to 70 Mbits/s, `fyaf` is not a bottleneck that affects the performance evaluation results significantly.

Figure 7.3 The relative throughput of `fyaf` for a wide range of network loads.



7.6.3 Preliminary Task 2: Overhead of monitors and TG

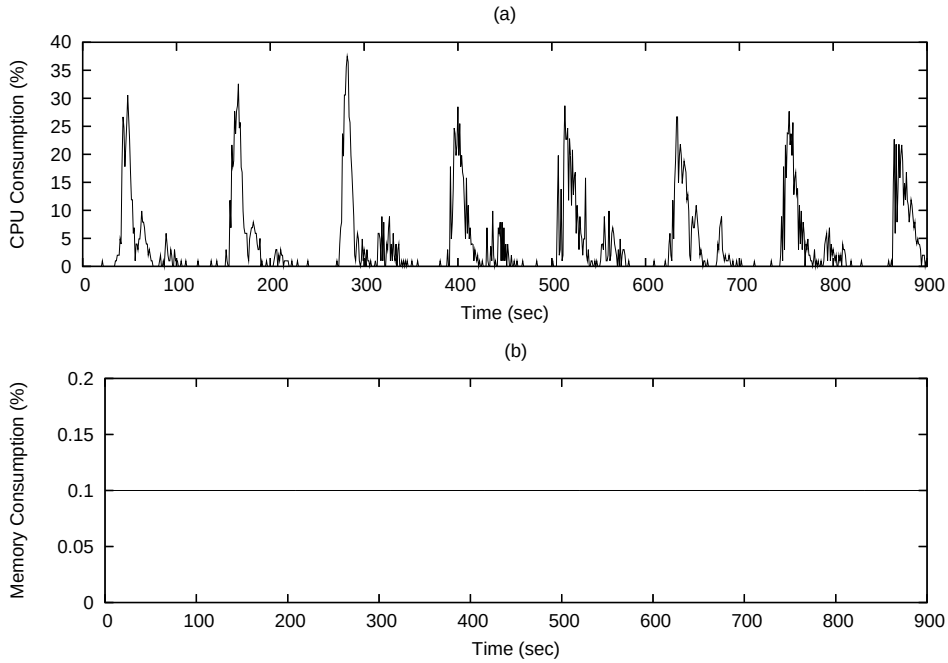
Monitors

As mentioned in Chapter 6, the monitors are not considered to add a significant overhead to the running of the tasks. Though, we have to verify this more explicitly. We approach this by selecting one of the several runs from the tasks, and investigate the `top` results from that run. We argue for its representability by investigating several other runs. Thus, as stated in Chapter 6, the monitors run passively in the background, a behavior which implies that we do not expect any dependency with regard to the factors introduced earlier in this chapter. The results are based on the one-hour experiment shown in the Task 2 discussion. The data is obtained from the third run. Table 7.1 shows the average results. There are some processes that are numbered, e.g. “experiment_client1” to “experiment_client6”. This is because the scripts fork during the run-time. As seen, the overall overhead for the monitors and scripts do not play a dominating role in the evaluation. The load is relatively constant, as well, i.e., we do not observe any significant peaks in the data files, as shown in the rows displaying the maximum values.

pid	process	μ CPU	Max CPU	μ RAM	Max RAM
13139	sscript	0.0000	0.0000	0.1135	0.2000
13207	super_script1	0.0000	0.0000	0.2000	0.2000
13214	super_script2	0.0000	0.0000	0.0000	0.0000
13752	super_script3	0.0000	0.0000	0.0000	0.0000
14328	experiment_client1	0.0000	0.0000	0.2085	0.3000
14329	experiment_client2	0.0000	0.0000	0.2219	0.3000
14330	experiment_client3	0.0000	0.0000	0.2095	0.3000
14628	experiment_client4	0.0000	0.0000	0.2086	0.3000
14630	experiment_client5	0.0000	0.0000	0.2085	0.3000
14637	experiment_client6	0.0000	0.0000	0.2085	0.3000
14632	tg_server_run1	0.0000	0.0000	0.1079	0.2000
14635	tg_server_run2	0.0000	0.0000	0.1079	0.2000
14629	vmstat1	0.0000	0.0000	0.1000	0.1000
14639	top1	0.6626	2.0000	0.1000	0.1000
14638	top2	0.0000	0.0000	0.1000	0.1000
14631	sar1	0.0000	0.0000	0.1000	0.1000
14634	sadc1	0.1336	2.0000	0.1000	0.1000
14640	fyaf	0.0000	0.0000	0.1000	0.1000
	Sum	0.7962		2.3948	

Table 7.1: Consumtion of scripts and monitors.

Figure 7.4 Consumption of TG's TCP server.



TG

Table 7.1 shows two tg_server_run processes. These are the scripts that start the TCP server, which is a separate process. Naturally, as the server receives more data and is an active part of the data transfer, its process consumes more resources correspondingly. TG stores packet statistics to file, thus causing some disk I/O overhead. We choose to focus on plotting the first 15 minutes of the same run as mentioned above. Figure 7.4 shows the results. The CPU consumption is shown as (a) in the figure. We get an average of 2.8140 % for the CPU utilization, but the maximum utilization is 46.8 %. In (b) in the figure, we see that the memory utilization is constant over the same time interval, thus we focus on the CPU consumption. These CPU peaks may play a role in the overall performance of TelegraphCQ. The peaks appear each second minute, and we can not identify the cause of this behavior.

Introspective	Not introspective	π from `streams.iptcp`
`task_101`	`task_101.3`	*
`task_101.1`	`task_101.4`	`sourceip`, `destip`, `sourceport`, `destport`
`task_101.2`	`task_101.5`	`destip`

Table 7.2: A mapping of tasks and projections in Preliminary Task 3.

7.6.4 Preliminary Task 3: Overhead of introspective TelegraphCQ streams and verification of shedded tuple streams

We have inserted the introspective TelegraphCQ streams[2] to help us investigate queries, operators and queues. These streams may also provide a considerable overhead, since - especially - `tcq_queues` constantly reports about the Fjords queues and modules. This preliminary investigates the overhead. Fortunately, we can estimate the overhead of the load shedding, i.e., the Data Triage tuples, as mentioned in Section 4.3.1, and the accuracy at the same time. The load shedding and accuracy should behave relative to each other, i.e., if TelegraphCQ reports a packet loss of 100 packets over a given time window, the reported number of packets should be 100 less than what `fyaf` sent.

Setup

The sub-tasks in Preliminary Task 3 are named `task_101` to `task_101.5`. All six sub-tasks perform simple projections that select from the `streams.iptcp` stream. Table 7.2 shows the sub-tasks and which attributes they project. The number of projected attributes is reduced for each of the three query types, but still all tuples are selected. The first three sub-tasks use the introspective queries, while the last three do not, where we investigate at what network load TelegraphCQ starts dropping packets.

We firstly investigate the overhead of the introspective queries. We calculate the relative throughput and compare the results.

The second experiment investigates the reliability of the projections. Corrected by the reported relative throughput, we compare the number of packets written by TelegraphCQ, to the number of packets sent by `fyaf`.

Table 7.3 shows the parameters for the pre-tasks, since they differ slightly from the parameters given earlier in this chapter.

[2] The introspective streams are introduced in Section 4.3.3 and their stream definitions are located in Appendix C and in [Dyn].

Run time	# seconds
Client	60
Server	90
Number of runs	10
Reference to error-tables	D.2
Network load (Mb/s)	1, 5, 10, 20, 25, 30, 40

Table 7.3: General overview of the introspective tasks.

Results

Figure 7.5 shows the result from running the six sub-tasks. The greatest gap between the curves appears at 10 Mbits/s, and we see that the introspective sub-tasks generally have a lower relative throughput than the non-introspective tasks, as expected. The result is also shown in the center column in Table D.1. At 30 Mbits/s, the relative throughput is almost zero. We conclude that at simple projections, and at the system used in this performance evaluation, TelegraphCQ does not manage a rate higher than 30 Mbits/s.

The second data analysis results are shown in Figure 7.6 and Table D.1. The number is TelegraphCQ's result subtracted from `fyaf`'s, i.e. the system's result, such that a zero means the two parts give equal results. As mentioned in the previous section, the reported relative throughput is corrected for, so that when TelegraphCQ reports a relative throughput of 50 %, this is added to the result. We see that both `task_101` and `task_101.3` stand out compared to the remaining tasks. Though, amazingly, if we exclude `task_101`, we see that the non-introspective tasks report a higher relative throughput than what is the case. For instance, `task_101.4` at 10 Mbits/s reports a relative throughput of 79.824 %, while the system reports a relative throughput of 72.788 %. The corresponding `task_101.1`, with the introspective queries included, reports a lower relative throughput, which is also more correct, according to the system results.

The only main difference between the two types of tasks is that the introspective `tcq_queues` uses resources since it constantly obtains queuing information from TelegraphCQ and writes this information to file. Though, `task_101` shows a significant error with regard to these conclusions. As mentioned in Chapter 4, the introspective queries are not yet fully included into the DSMS. This means that there might be some problems in the coordination between the wrapper clearing house (WCH) - which receives the packets and reports packet loss - and the internal functions for the introspective queries. Another explanation is that if the introspective query adds an overhead to the system, the WCH drops more tuples, giving more time for the eddy to send the tuples to the output. This is a somewhat vague explanation; the queries perform simple projections, which should only add a minimal overhead to the processing. The eddy probably sends the packets to only

Figure 7.5 Relative throughput for tasks using introspective queries and tasks not using introspective queries.

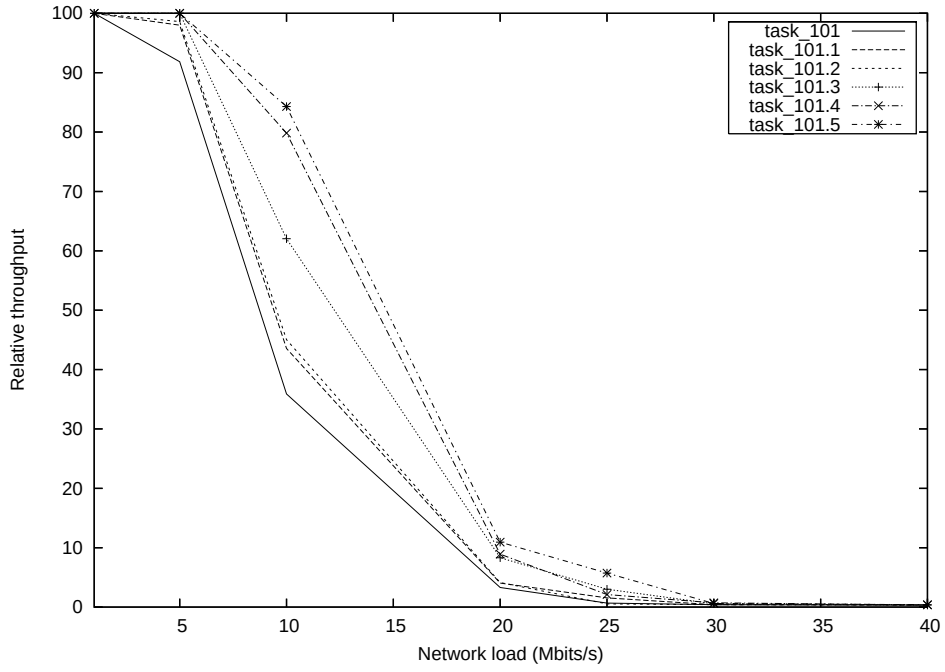
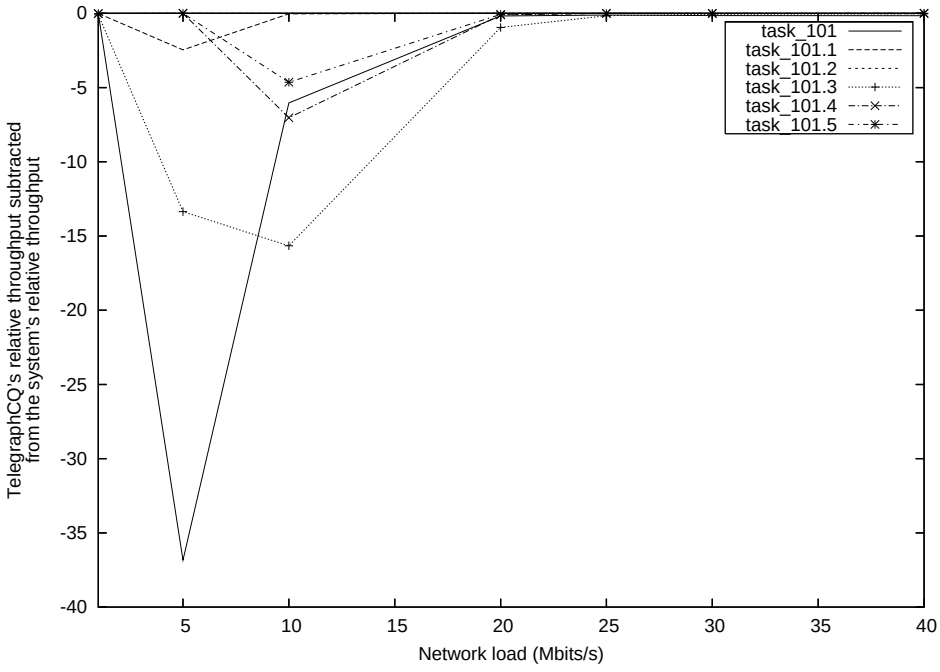


Figure 7.6 The difference between the system's and TelegraphCQ's report of dropped packets.



Sub-tasks	`task_1.1`, `task_1.2`, and `task_1.5`
Reference to error-tables	E.1
Network load (Mb/s)	1, 2, 2.5, 5, 7.5, and 10

Table 7.4: General overview of Task 1.

one filtering module. `tcq_operators` only informs that there has been created three Fjords scan modules for example 5 Mbits/s for `task_101`. The scan modules obtains data from the shared memory, and the three modules probably obtain the tuples from the two Data Triage streams and `streams.iptcp`. Thus, we can not use `tcq_operators` to inform about the filtering modules; we have to assume this based on the theoretical background presented in Chapter 4. Though, still it is challenging to understand why the results are so unexpected at 5 Mbits/s. This is considered a task for future work.

Without going deeper into these results, we may consider that using the introspective queries may give correct results as long as the queries do not project all attributes in the tuple. We also see that at 10 Mbits/s, the introspective queries have a relative throughput of approximately 40 %. Thus, we keep a network load of 10 Mbits/s as the upper limit for the following tasks.

7.6.5 Task 1: Measure the average load of packets and network load per second over a one minute interval

Setup

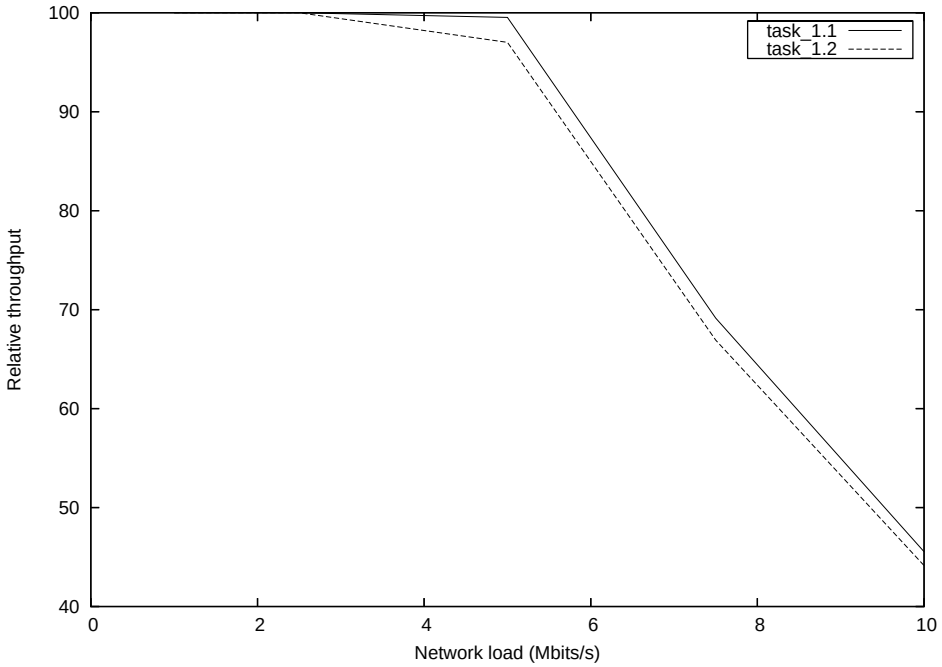
Task 1 investigates the DSMS's ability to report the average number of packets and average network load each second over a one minute window. In Chapter 5, we show two alternative solutions on how to solve the task; one using two queries and the other using only one query.

Our first approach is to run the two alternatives separately and compare the relative throughput. The task having the highest relative throughput is investigated with regard to accuracy.

The second approach is to run both the alternatives concurrently and comparing the differences between these results and the results from the first approach. We focus on the relative throughput and accuracy from the first approach and compare both on these metrics.

Table 7.4 shows the general overview of the conditions in Task 1. These settings are used in all the sub-tasks in Task 1. `task_1.1` and `task_1.2` corresponds to the first approach, while `task_1.5` corresponds to the concurrency run.

Figure 7.7 Relative throughput for `task_1.1` and `task_1.2`.



Results

We start by investigating the relative throughput. As defined earlier, the relative throughput describes the packet dropping as reported by TelegraphCQ. As for Task 1, the queries focus on relative throughput, as well, so both relative throughput and accuracy metrics are discussed as one.

Note that we generally display graphs in the discussion and tables in the corresponding appendix. Figure 7.7 and Table E.2 show the results from the TelegraphCQ relative throughput. The two tasks follow each other almost perfectly, though `task_1.1` has a higher relative throughput than `task_1.2`. This is somewhat unexpected, since `task_1.1` uses a sub-query to obtain the tuples, an action that would indicate more calculation, thus lower relative throughput.

What we assume is that since every tuple has to visit the eddy, the usage of a sub-query may increase the total load for the eddy, and thus decreases its efficiency. But, as the modules for the aggregations in the `streams.task_1.1` creation probably return tuples to the eddy at a very low rate[3], the final query only needs to handle 60 tuples, which is a small number compared to the possible rate.

As mentioned in Chapter 5, `task_1.2` does not perform any additional calculations. Hence, we only calculate the relative throughput from `streams.iptcp`.

[3] Due to the one second window length.

The windows are filled with tuples for one minute, and all the tuples are calculated each second. This may explain why `task_1.2` reports a lower relative throughput than `task_1.1`.

The final assumption is that tuples may be shedded inside the eddy. This is really not possible, since this functionality does not seem to be supported. Based on how the architecture is presented in the literature [RH06, RH04], we conclude that only the WCH reports about shedded tuples. This implies that streams within the `WITH`-clause are not registered by the WCH, and thus do not give any results about shedded tuples. We have experienced this behavior in tests where we insert drop statistics to streams only used internally. Nothing is displayed in the result files. This also means that using many sub-queries may overload the eddy extensively, giving strange behavior, unless the rate is low, as in `task_1.1`.

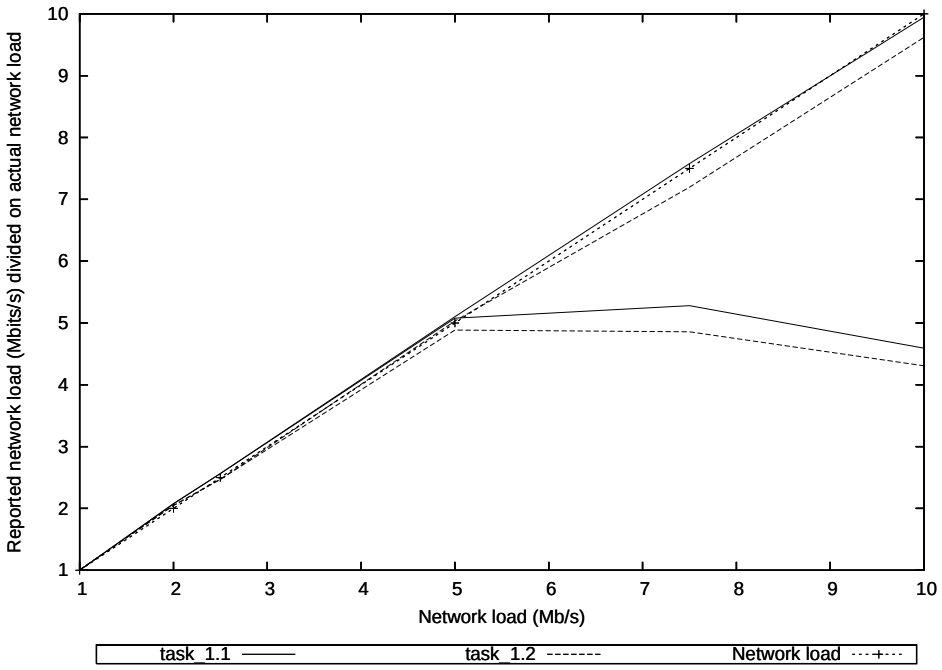
Since we can not say anything more detailed about the TelegraphCQ internals at the time, we investigate the results from the queries. We investigate the number of reported packets and network load to see if there are any significant differences in the two results. Note that we have to change the results from `task_1.2` such that the results correspond correctly. We multiply the second attribute with the first one, and finally multiply the product with eight to get the bits. Table 7.5 shows the difference in network load and average number of packets between the two sub-tasks. As mentioned in Chapter 5 the network load is calculated by the `totalLength` IP field, omitting the Ethernet packet. When TG sends packets at 1 Mbits/s, the Ethernet header is included. This means that we have to correct for this as well. Except from 1 Mbit/s, we see that `task_1.1` reports a higher network load than `task_1.2`.

The corrected results - with respect to the Ethernet header - are shown in Figure 7.8. The network load is shown as a linear relation between the two axis. A correct result maps directly on the dotted line named *Network load*. By dividing the output with 1 Mbits, i.e., 1,048,576 bits, and adding the Ethernet header, the figure apparently shows that both `task_1.1` and `task_1.2` are correct to begin with, but start to spread significantly from the actual network load at 5 Mbits/s. The inaccuracy observed at rates higher than 5 Mbits/s corresponds to the drop of relative throughput shown in Figure 7.7, thus implying that TelegraphCQ's accuracy is still high, as long the relative throughput is correct. This is shown by adjusting the results to the relative throughput. As we see in Figure 7.8, `task_1.1` seems to follow the actual network load more correctly than `task_1.2`.

One issue to be considered is that the results from the set of network loads are based on the average of the results from the total run. If we investigate Figure E.3 and Figure E.4, we see that the stream for `task_1.2` shows that the stream is turned off earlier than for `task_1.1`.

Though, `task_1.1` - as we see from the two figures - has curves that are similar to `task_1.2`. This means that we choose to ignore the calculation of the average as relevant for the differences we have experienced. A limitation of the range in

Figure 7.8 Comparison of network load reported in `task_1.1`, `task_1.2`, and actual network load.



which the average was calculated might have given an even more accurate result.

We see that neither of the two tasks report a network load that is exactly correct according to the expected rate. As mentioned earlier in this chapter, this may be because of TG's inter-arrival rate or the TCP layer's manipulation of the packets. Sending packets at 1 Mbits/s does not mean that exactly 1,048,576 bits pass through the NIC each second.

To investigate these issues, we need to look at the network load reported from `sar`. `sar` reports both the number of packets and number of bytes sent and received at the NIC. Since `fyaf` obtains packets directly from the NIC, `sar`'s results may give some further hints about the accuracy. Though, as the results so far have shown that TelegraphCQ reports results that are relatively accurate to the relative throughput, we choose to include further investigation with regard to `sar` output to future work.

Based on the discussion above, we have decided to focus on comparing the results from `task_1.1` with a concurrency test. This is because `task_1.1` seems to give the most accurate reports. The concurrency is tested in `task_1.5`, which runs both `task_1.1` and `task_1.2` concurrently. Table E.4 shows the relative throughput. Compared to the other tasks, as shown in Table E.2, we see that `task_1.5` has a lower relative throughput. This is expected, since both queries

(a) Network load.

Network load (Mbits/s)	`task_1.1`	`task_1.2`
1	1018653.378	1024751.958
2	2115193.834	2083975.662
2.5	2610418.442	2513309.212
5	5173026.783	4973607.457
7.5	5394644.746	4957755.047
10	4695717.436	4401112.718

(b) Average number of packets per second.

Network load (Mbits/s)	`task_1.1`	`task_1.2`
1	270.076	276.738
2	562.825	562.513
2.5	690.261	689.235
5	1351.474	1321.986
7.5	1273.005	1205.471
10	1062.979	1029.633

Table 7.5: Average of reported network load and number of packets for `task_1.1` and `task_1.2`.

are run concurrently, thus leading to more calculations, more modules, and more tuples for the eddy to handle. None of the queries are similar, and thus do probably not help the CACQ to optimize the modules.

If we look at the reported network load - as seen in Table 7.6 - we observe a curve similar to what `task_1.2` reported in Table 7.5. A comparison, as seen in Figure 7.9 also verifies this statement, by showing that, at rates where the relative throughput starts to fall, `task_1.2` and `task_1.5` seem to give similar results. But, at 10 Mbits/s, the difference starts to be significant.

The results from Task 1 shows that there is a possibility of using TelegraphCQ for tasks where simple aggregation is used to extract information about the rate of the flow. Still, as we see in Preliminary Task 3, the data rate TelegraphCQ supports is relatively low, given our stream parameters. Thus, the shedding information can be considered accurate, as seen in Figure 7.8. Thus, it is possible to follow up the ideas posed in [RH06] to see how the results from the shedded tuples' queries actually can be integrated into the result stream to adjust for the errors.

(a) Network load.

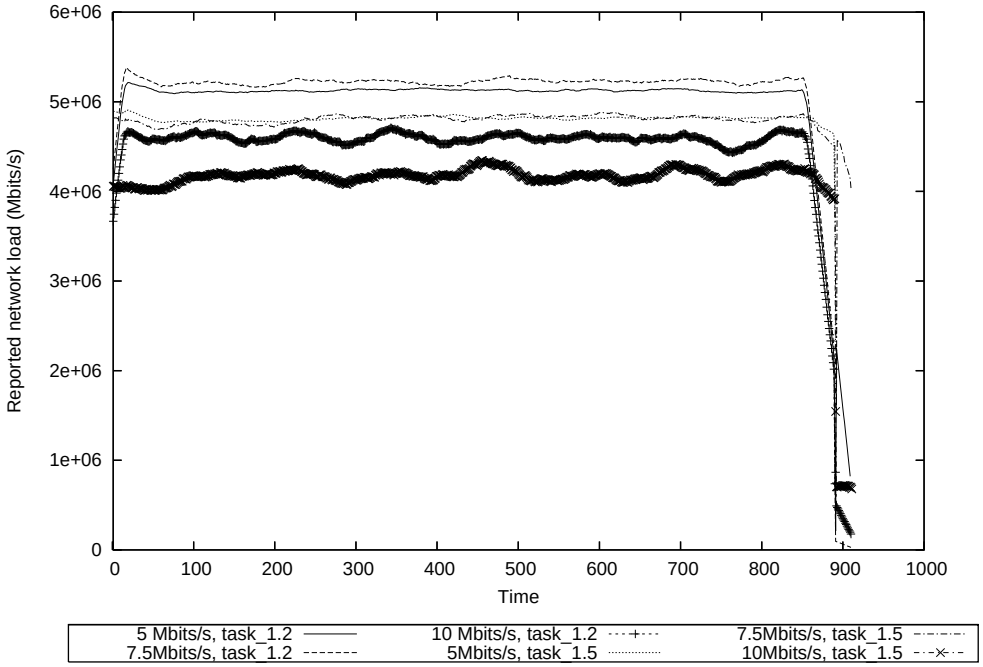
Network load (Mbits/s)	`task_1.5`
1	1054975.126
2	2111964.958
2.5	2587628.847
5	4810705.974
7.5	4788337.374
10	4093045.231

(b) Average number of packets per second.

Network load (Mbits/s)	`task_1.5`
1	277.698
2	554.998
2.5	679.748
5	1260.151
7.5	1115.060
10	932.502

Table 7.6: Average of reported network load and number of packets from `task_1.1` as reported in `task_1.5`.

Figure 7.9 Comparison of network load reported in `task_1.2` and `task_1.5`'s version of `task_1.1`.



Sub-tasks	`task_2.1`, `task_2.2`, and `task_2.3`
Reference to error-table	F.1
Network load (Mb/s)	Varying between 1 and 10.

Table 7.7: General overview of Task 2.

7.6.6 Task 2: How many packets have been sent to certain ports during the last five minutes?

Setup

As stated in Chapter 5, Task 2 aims to measure joins between a stream and a table. The performance is based on three table sizes, using one sub-task for each table size:

- *65,536 ports*. This is the total range of ports available in TCP. This would possibly lead to a large overhead for searching through parts of the table, had it been implemented as B-trees, but as the SteMs are implemented using hash indexes, we assume that the lookup is fast. Even though, as this range may not be applied in the real life monitoring - as stated in [ian] and Chapter 5, some ports are more interesting to investigate than others - our experiments investigate the worst case; all ports are interesting. `task_2.1` uses this table size.
- *1,026 ports*. This number emulates a search through the dedicated ports. The setup forces us to use port 60,011 as destination port. This could actually be *any* port, so we assume the results to be independent of these restrictions. `task_2.2` uses this table size.
- *1 port*. This is simply a base test. We do expect to see the effect of a single lookup, i.e., identifying the overhead which is constant at a lookup. `task_2.3` uses this table size.

Ideally, we expect no differences in the results between the three tables, since the hash lookup generally is considered to be $O(1)$. The first table matches packets heading for both `dmms-lab65` and `dmms-lab60`.

The additional parameter information is given in Table 7.7.

As with Task 1, we run all the three tasks at up to 10 Mbits/s and investigate the relative throughput. Since we use sliding windows of five minutes, we will probably observe that the relative throughput changes at higher rates when TelegraphCQ starts printing the results and calculating the windows. To investigate the accuracy, we investigate the results from `sar`. As stated in the discussion in Task 1,

`sar` reports the number of packets received each second, thus, since we use windows of five minutes, we have a possibility of approximating the expected result. This task shows the possibilities of using the system monitors for investigating the consumption metric.

Results

Table F.1 shows that Task 2's results are characterized by more errors than Task 1. As mentioned earlier in this chapter, we only focus on tasks that have some error-free runs. Thus, this may affect the credibility of the results. The fields marked as "not tested" denote that the sub-task/network load-pair was not run at all, since the network loads tested varies.

For the first sub-tasks, Figure 7.10 and Table F.3 show that `task_2.1` - probably mostly because of its two hits - stands out from the two following sub-tasks, which do not differ significantly from each other. We assume this is because of the hash lookups and one hit of matching ports. To save space, we choose to let `task_2.3` represent both itself and `task_2.2`. In Task 1, we show the relative throughput by calculating the average over the whole run. As shown in e.g. Figure 7.9, we see that the whole run reports an approximately even result throughout the whole run. As we see in the following results, this may not be the case in Task 2. Thus, the result in Figure 7.10 does not give a fair view of the total run. As for Task 1, we need to show the relative throughput over the time sequence of 900 seconds to get a proper overview.

Figures 7.11 and 7.12 show how the relative throughput for `task_2.1` and `task_2.3` respectively.

For `task_2.1`, only the 1 Mbits/s runs manage to keep a relative throughput of 100 % throughout the whole run. For `task_2.3`, the 2.5 Mbits/s run keeps a maximum relative throughput as well. This network load is not represented in `task_2.1`. The figures show that the relative throughput drastically drops when the calculations start after approximately five minutes. Though, we see that TelegraphCQ increases the relative throughput once more before dropping again after five new minutes. This behavior is somewhat interesting with respect to the fact that we use sliding windows. An immediate impression would be that the relative throughput would be constant, since we assume that TelegraphCQ recalculates the window for each second. Since the query uses a `COUNT(*)`, TelegraphCQ may for example have distinct count for each second of running and remove this count from the total count while adding the values for the new second. An optimization would possibly be to perform distinct counts in sub-queries, as done in `task_1.1` and proposed in Task 6.

We see a peek after 600 seconds. This may be explained by a manageable amount of tuples in the window, since the relative throughput has been generally low for five minutes. This manageable amount probably makes it easier for the eddy to

Figure 7.10 Relative throughput for Task 2.

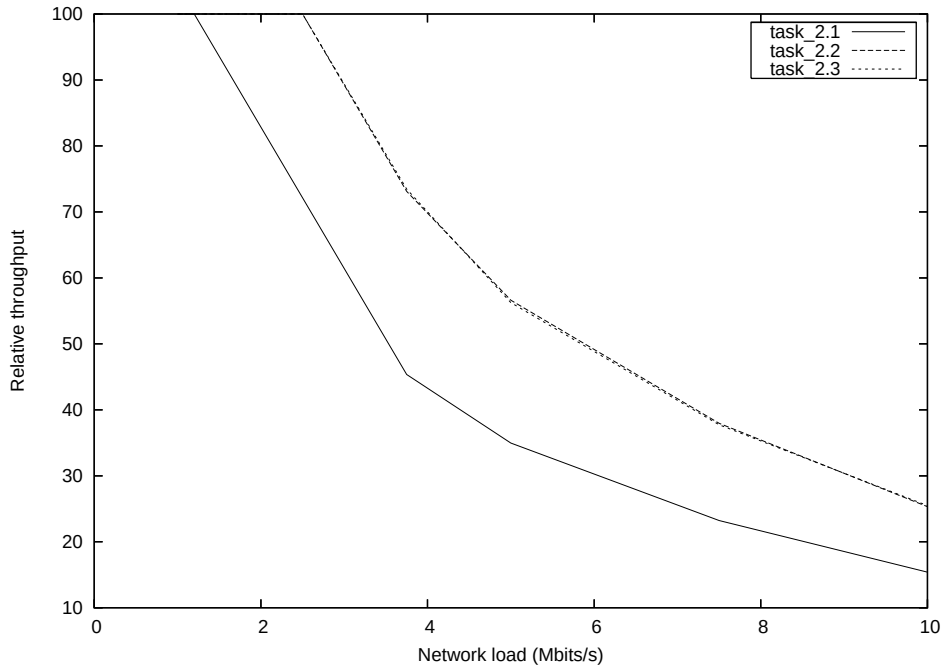


Figure 7.11 Relative Throughput for `task_2.1`.

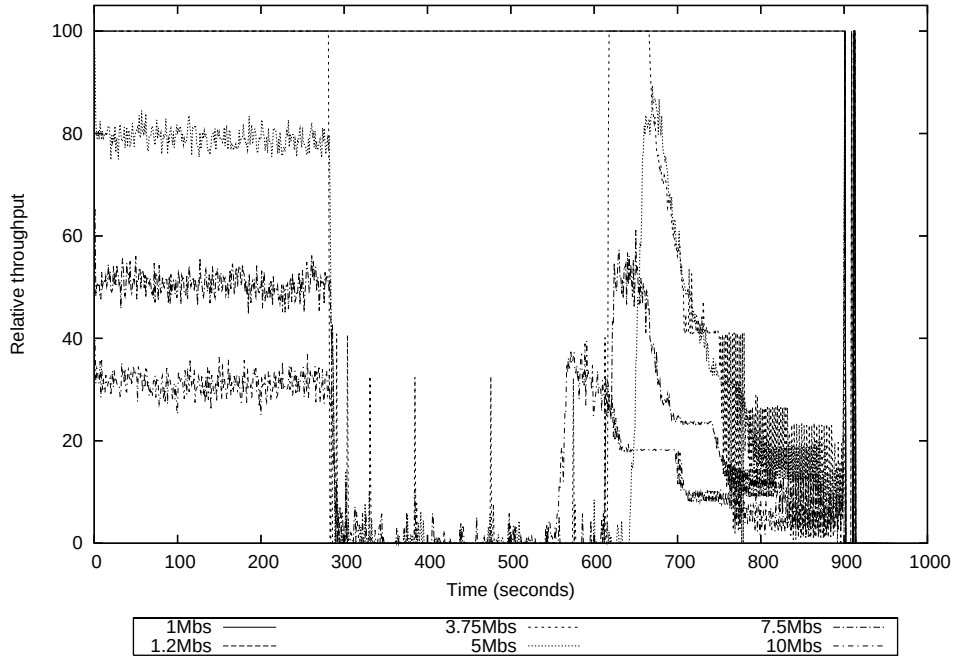


Figure 7.12 Relative Throughput for `task_2.2` and `task_2.3`, as represented by `task_2.3`.

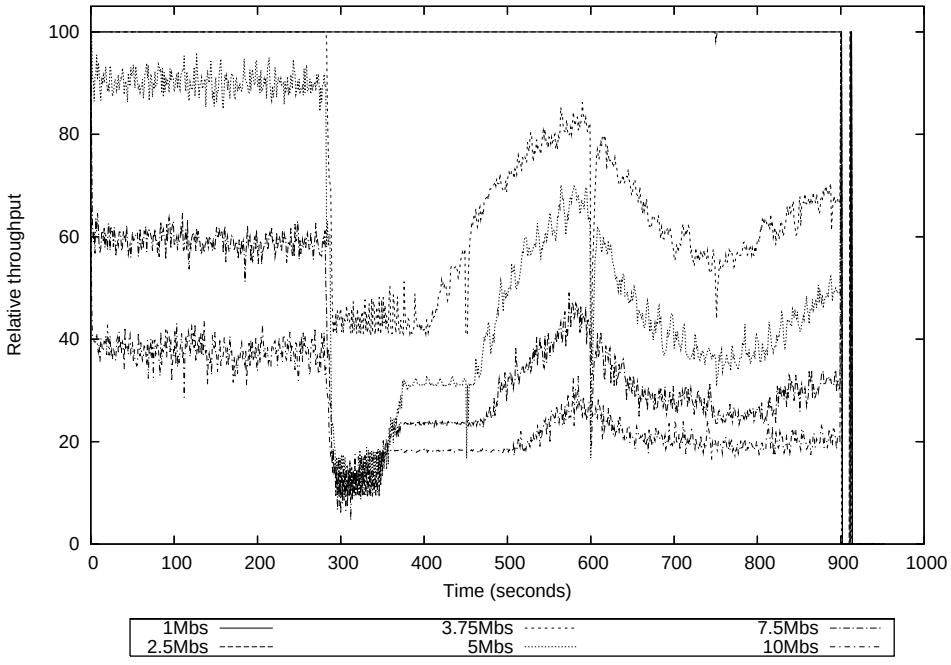
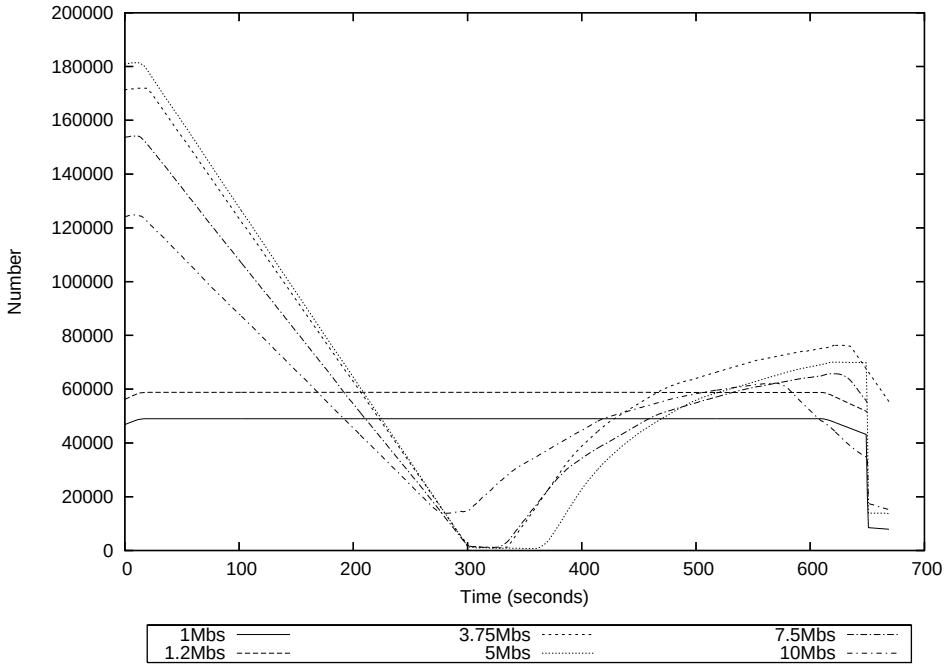


Figure 7.13 Number of packets destined for `dmms-lab65` in `task_2.1`.



obtain more tuples from the shared memory, thus informing the wrapper that it does not need to drop all the tuples it receives.

If we investigate the results from `task_2.1` and `task_2.3`, as shown in Figures 7.13 and 7.14, we see that the output starts at approximately 255 seconds after the query starts[4]. Since TG sends at a constant rate, we know that the expected result is only supposed to be correct as it is represented in the 1 Mbits/s run. The decreasing of the result curve is a natural consequence of the sudden drop of packets; when the relative throughput reaches e.g. 0 %, this immediately affects the results, as we see.

To verify that the results are correct, we investigate the `sar` results. We focus on the third 1 Mbits/s run, and the results from five to ten minutes. The script that created the `sar`-based result file summed up all the results for the last 300 seconds for each new second. As we see in Figure 7.15, the accuracy is considerable. The main difference is the first 25 seconds of the curve. But, as noted in Chapter 6, `sar` may leap over a couple of seconds when it reports the system activities, thus giving such results in some occasions. We have not focused on adjusting these errors, even though it is manageable.

[4]Note that the calculation of the relative throughput starts at once the query is started; we subtract the query results' end time from the relative throughput results' end time (server time) to find the start time for the output. In this task, it means that even though the results start at time 0, it is not the

Figure 7.14 Number of packets destined for `dmms-lab65` in `task_2.2` and `task_2.3`, as represented by `task_2.3`.

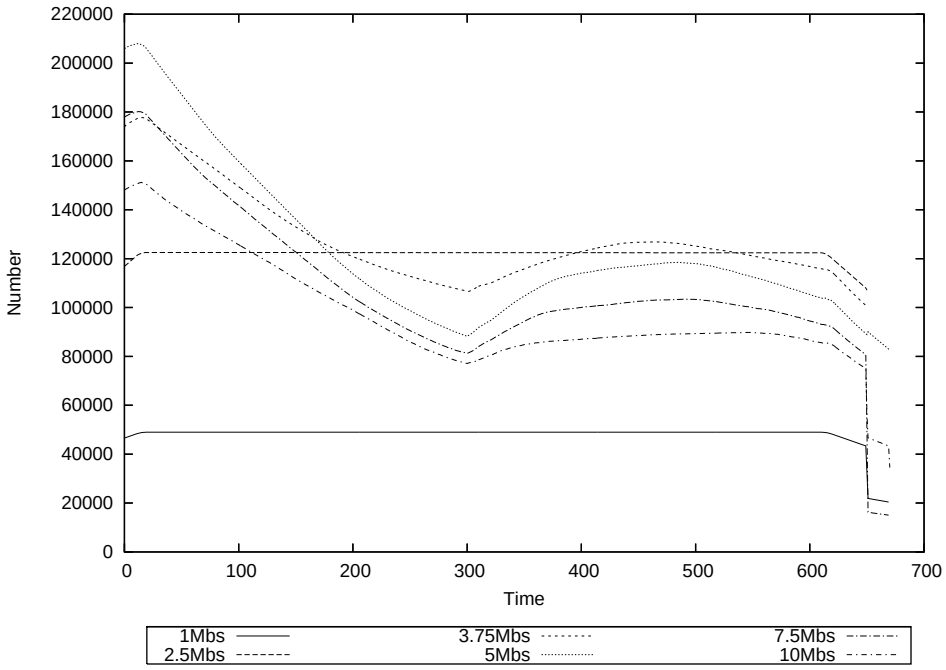
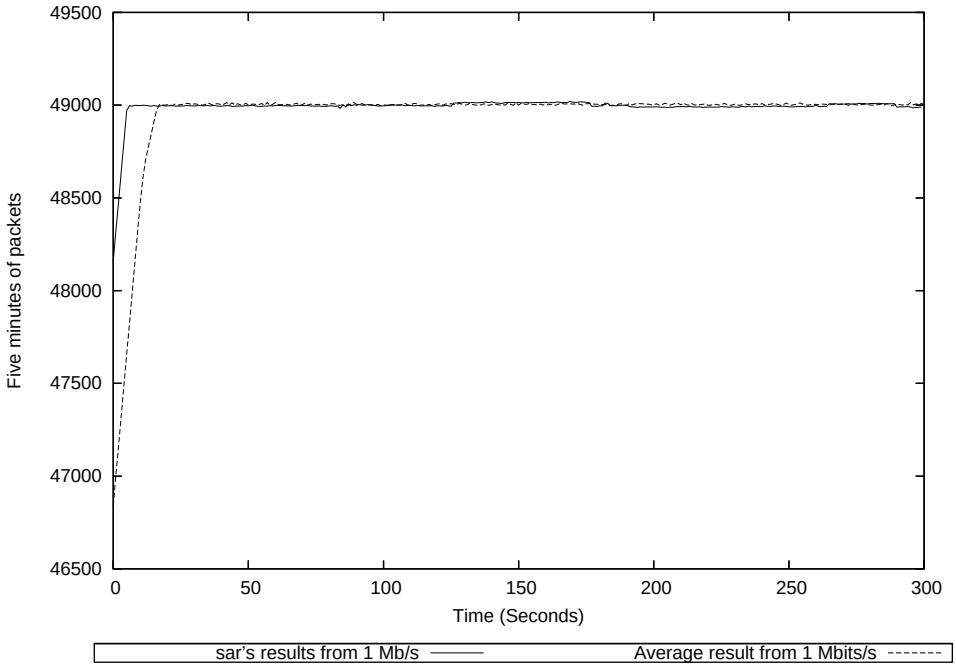


Figure 7.15 Accuracy of number of packets reported.



Still, the results seem unexpected. We see that the drop after the 600 second's pike is not as definite as the first drop. This may be caused by several factors aside from TelegraphCQ's windows. A possibility is that some system settings have changed over the time period. Another is that the eddy has performed an adaption.

We explore the first possibility by further investigating the output from `sar`. Since we see a similar pattern in each run, we choose to explore the median run at 5 Mbits/s for `task_2.1`, i.e., the third run. As for the accuracy test, we assume that the third run assembles the average of `task_2.1` in a satisfying manner. As seen in Figure F.2, they almost map one-to-one. Note that the information gathered by `sar` is not perfectly correct and thus only shows some trends in the system.

Figure 7.16 shows that there is a slight drop in both `user` and `system` CPU utilization when TelegraphCQ starts the output. Still, the memory usage increases and reaches a peak after ten minutes. Compared to the results reported from the sub-task, we see that the number of reported packets is inversely proportional with the memory usage. We see that the reported number of packets increases after five minutes of output. This resembles the peak in the reported memory usage.

So far, we can not conclude that the system monitors give any results revealing TelegraphCQ's result; they seem to reflect TelegraphCQ's behavior, but we do not know which affects which. Had the CPU utilization increased after five minutes, we could have easier concluded that there was a resemblance between the system and TelegraphCQ.

If we look at the introspective streams, especially `tcq_queues`, we may reveal the reason for, e.g., the sudden increasing of reported packets after five minutes of output. We still focus on the same run as above.

The `tcq_operators` stream informs that the two SteMs, `ports.port` and `iptcp.destport`, are identified by `opid`s zero and one, respectively. Thus, we investigate only these two operators, their I/O queues and kind. The table overview of the introspective streams is shown in Appendix C. As we see in Figure 7.17, no new information is revealed. The operators almost have equal values. Note that `tcq_queues` has not reported every second, so there may be unprecise results. Though, we see the trend: When the relative throughput starts to drop, the number of tuples handled by the current SteMs drops correspondingly.

With the available monitoring data, we may conclude that the results are affected by the increase of memory usage. As presented in Chapter 5, storage of five minutes of data at a network load of 100 Mbits/s may use approximately 10 Mbytes of memory. At 5 Mbits/s, this is probably not so much of a problem.

Though, we investigate the second possibility; that what we observe is the beginning of an adaption process that is a cooperation between the eddy and the wrapper. To investigate this, we run the sub-task for a longer period than 15 minutes. We run `task_2.1` at 5 Mbits/s, five times and one hour each run. Figure 7.18 shows

start time for the run.

the result from the output. The curve shows an harmonic reduction that seems to end at an average of 56,406.52 packets per five minutes. As we see, the harmonic curve may be caused by some adaptivity mechanisms. The dotted lines in the figure shows the result of a function trying to describe the behavior of TelegraphCQ. The modeled function uses five variables:

- x denotes the time in seconds.
- n is the expected number of packets if the relative throughput was 100 %. For 5 Mbits/s this is expected to be approximately 250 000. We base this approximation on the average result from the 1 Mbits/s runs.
- p is the period of the cosinus function. We set $p = 600$, since is seems like this is the period in the observed run. This probably corresponds to a double window size, since the windows in Task 2 is 300 seconds.
- a gives the amplitude of the function.
- c shows the number of packets convergence value. We set it to 56406.52 since this is the average mentioned above.

Based on these variables, we utilize an harmonic reduction function:

$$f(x) = e^{-\frac{x}{a}} cos\frac{2\pi}{p}(x)n + c.$$

As we see in Figure 7.18, the first period is longer than the next ones. This is not included in our function, thus making it not very reliable. Though, we observe regularities that may be described analytically.

We see that the result of 5 Mbits/s converges to an average of 56,406.52 packets five minutes. From this value, we can possibly estimate the maximum network load TelegraphCQ manages to handle in this sub-task. $56,406.52/300$ gives approximately 188 packets each second. As shown, the analytical modeling may help investigating the possibilities of predicting the results. This may be helpful when using TelegraphCQ for longer periods, as one may use functions to find out what number of packets TelegraphCQ converge to, and when it for example reaches a given confidence interval for this value.

Task 2 has shown the performance of joining a stream and a relation. As we discuss in Chapter 5, this feature is important for a DSMS, as it often is an integrated part of a DBMS. We argue that TelegraphCQ manages to output correct results when the relative throughput is 100 %. Though, as we observe, when the window is sliding at a size of five minutes, there is a possibility of low relative throughput, but after a while, the results are observed to stabilize.

Figure 7.16 System monitors for `task_2.1`.

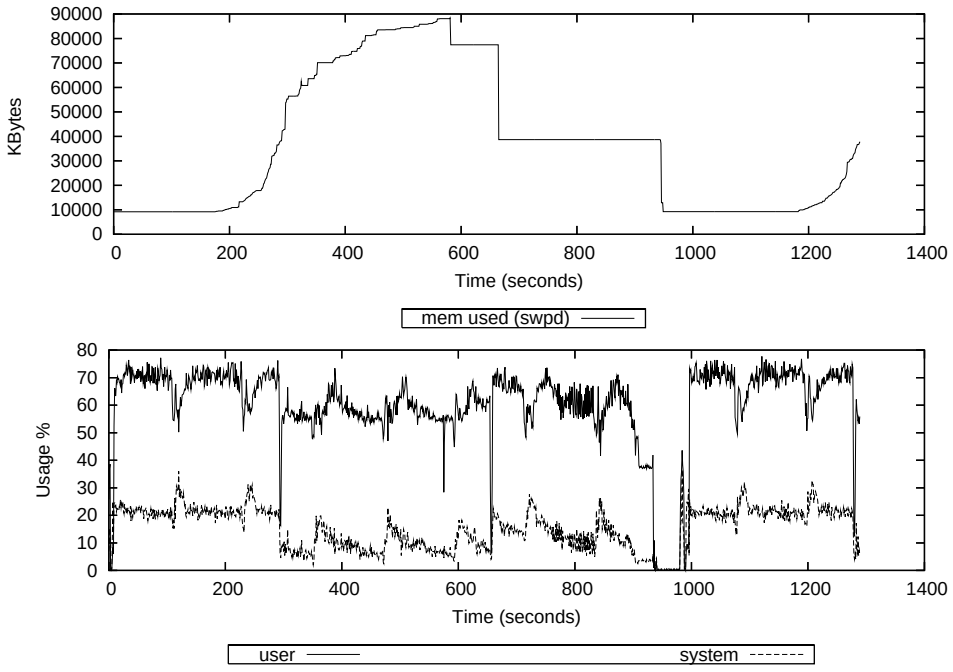


Figure 7.17 Introspective results for the third run at 5 Mbits/s, `task_2.1`.

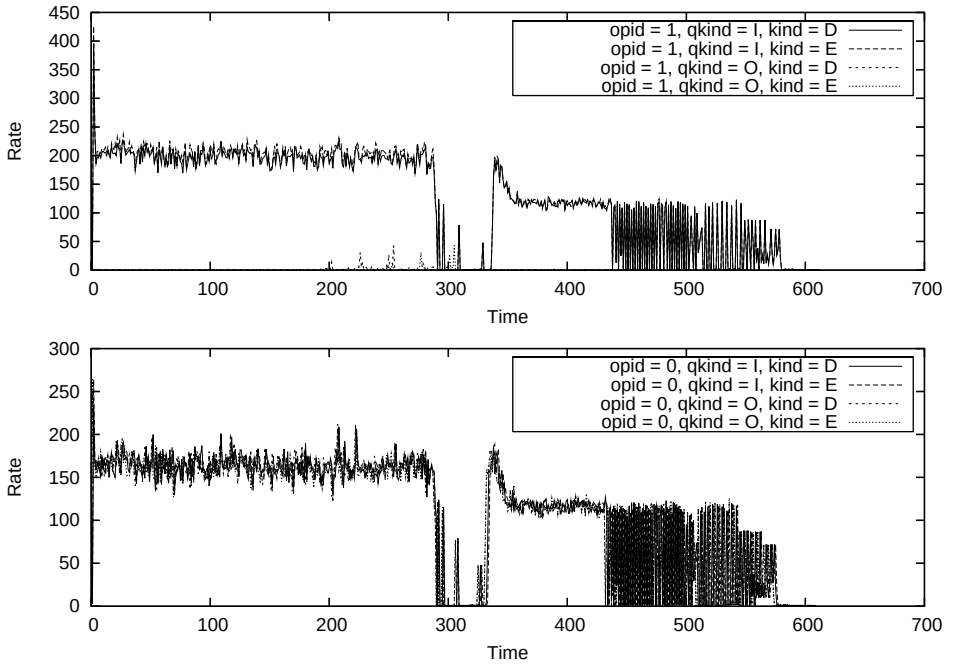
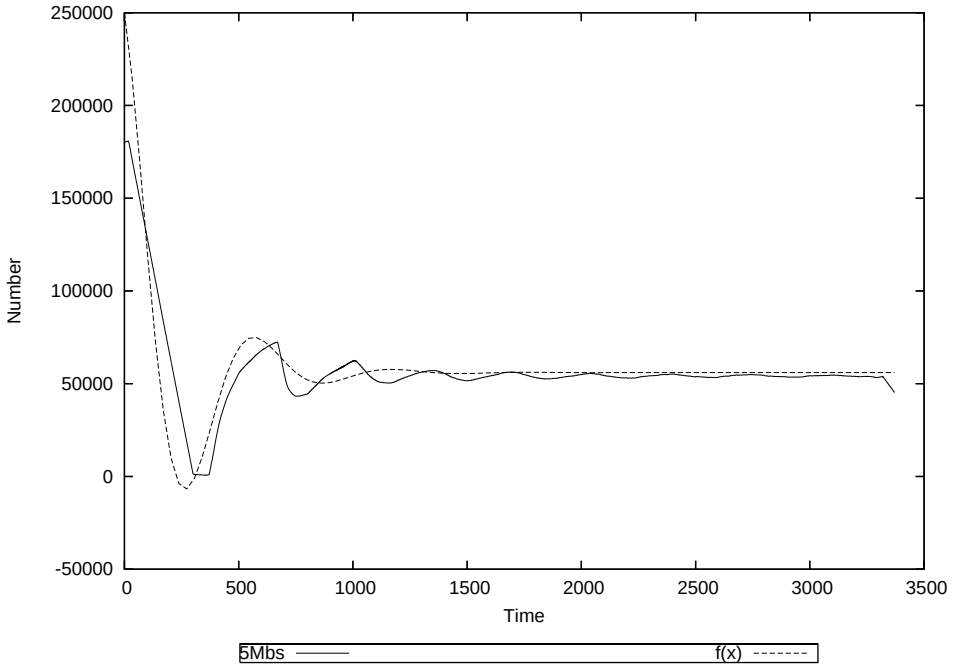


Figure 7.18 An hour of running `task_2.1` at 5 Mbits/s.



7.6.7 Task 3: How many bytes have been exchanged on each connection during the last ten seconds?

We assume that Task 3, as it is presented in Chapter 5, does not work properly since the tuples identifying the connections float out of the window at once the output starts. Thus, we investigate this task by showing the results and try to investigate why TelegraphCQ stops at once it gets a load too high for it to handle. As seen in Chapter 5 and Figures G.1 and G.2, and G.3, the query is rather complex, though still important to investigate due to its correctness criteria, which we argue for in the design of the task.

Setup

The setup uses on-line evaluation of ten TCP connections. The ten connections are established during the first ten seconds. The number of establishment also give a more realistic scenario than the single connection used in the prior tasks, since we assume that a link may be populated with at least ten connections.

Mostly, this setup is used to verify that it actually is the windowing scheme that is responsible for the incorrect results, and we try to show this by running the subtasks. Table 7.8 gives the additional parameter information.

Sub-tasks	`task_3.1`
Reference to error-table	G.1
Network load (Mb/s)	0.01, 0.1, and 1.

Table 7.8: General overview of Task 3.

`task_3.1` focuses on the connection identification and is slightly modified from the partial query shown in Figure G.1. The modification involves letting the `streams.conn` query be the main query. We focus on running `task_3.1`, since the rest of the task queries rely on these results.

Results

When we investigate the result from the three network loads at `task_3.1` we observe that the results are empty, i.e., we only observe a `success=0, cqcancel=0` output, which informs that the postmaster sends a message to the client, which it is turning off in a controlled manner. All the three network loads report a relative throughput of 100 %. As mentioned earlier in this chapter, we restart the postmaster for each run. Since we intend to start a new connection each of the ten first seconds, we assume that we observe at least nine connections at the first output.

Even though the task is verified by test streams as described in Appendix G.1, we have to investigate these results even further. Though, to verify that the query is correct, we insert two new queries in the stream that project all the tuples from `streams.syn` and `streams.synack`. The results are shown in Appendix G.2. We see that all the SYN and SYN/ACK tuples are registered. This indicates that some of the responsibility may lay in the main query. To investigate the main query, we rewrite the whole query so that it is now a PostgreSQL query. The reason for this is that we assume the only place where the main query can be incorrect, is at the conditions in the `WHERE`-clause. We do not join with the streams due to the sequence numbers. That the three streams, which are selected from, are all filling up their windows with the result tuples, strengthens our view. The location of the query and the result is shown in Appendix G.2. We see that the number and quality of the connections is correct. This probably means that there are some internal problems, which cause the final results not to appear.

Based on this result, we have to conclude that the connection identification does not work properly in TelegraphCQ. Thus, we end our investigations of Task 3 at this observation. We argue why in the following discussion, as we sum up the results and give an overview of the performance evaluation.

7.7 Summary of the Performance Evaluation

Even though there are design issues in Task 3 that could have been investigated even further, the results from the performance evaluation of the prior tasks have shown that TelegraphCQ only manages to handle data streams at approximately 2.5 Mbits/s having a constant rate and TCP payload of 576 bytes. Based on these results, we see that some kind of sampling is necessary to keep the results accurate. We know that TelegraphCQ performs this sampling on its own, by providing the possibility of querying the number of shedded tuples. The challenge with such a sampling is to know which tuples to drop and which to keep.

In the case of Task 3, we see that the result is based on the correct obtaining of three tuples; SYN, SYN/ACK, and the corresponding ACK. If one of these three tuples are shedded, and there are no re-transmissions, we do not manage to identify this particular connection. Thus, we see that even if TelegraphCQ manages to insert the connection tuples into a table, and even delete them, the probability of identifying the connection is very small. This implies that much of the packets in the network may not be registered with regard to payload, thus making the query, as we have proposed it, inaccurate by default.

We might have continued to investigate the query by looking at the payload queries. Though, because of the reasons mentioned above, we do not consider this as interesting. If we generalize the queries, we also see that what they perform is relation lookups. As noted, both relations and streams are stored in the SteMs as hash tables, and the stream-to-relation join is covered and discussed in Task 2.

If we generalize this discussion, at least another application mentioned in Chapter 2 and Chapter 5 is affected; the SYN flood DoS attack identification in Task 10. We already argue that TelegraphCQ does not manage to handle this task efficiently, but if it did, it still had to identify SYN/ACK and ACK packets. This means that we run the same risks as when the 3-way handshake is identified.

An alternative would of course be to investigate the number of packets that are SYN/ACK and number of packets that are ACK, as mentioned in Task 9. Though, since we do not know anything more than an attack has started, the task seem to be unhelpful[5] except from identifying that there has been an attack.

The performance evaluation has thus shown us that even though TelegraphCQ is accurate, it manages only very low network load. As we have noted throughout the thesis, this may be because of the considerable responsibility the eddy has while routing tuples.

As our evaluation is based on black-box testing of TelegraphCQ, we can not conclude that the eddy is responsible for the low relative throughput. Since the eddy is a module, the back end may create several eddies that have responsibilities of

[5] Turning off the server because there is a SYN flood attack on it only helps the persons initiating the attack.

different tuples and send the data to each other. In that case, Preliminary Task 3 has shown that, for simple projections, the relative throughput is low. Still, we assume that only one eddy is present in Preliminary Task 3.

The worst-case scenario we have used for the performance evaluation can have caused the results as well. Though, as described in [CCD+03] and Chapter 4, it has been discussed whether or not the eddy's tuple based routing should be considered using e.g. batching tuples that informs the eddy about major changes in the data stream's characteristics.

Chapter 8

Conclusion

This thesis shows the *design*, *implementation*, and *evaluation* of network monitoring tasks with the TelegraphCQ DSMS. We show a set of network monitoring tasks that we design and implement according to TelegraphCQ's query language. Some of these tasks are used in a performance evaluation of TelegraphCQ, as well.

In this chapter, we focus on each of the three different elements, but conclude the design and implementation in the same section, as in Chapter 5. Finally, the performance evaluation is discussed. Given the structure in the performance evaluation chapter, each task is ended by pointing out some concluding remarks. The conclusion in this chapter gives an overview of the results.

We also investigate the contributions of this thesis; as a summary of our work in the general network monitoring and analysis perspective. A section with a critical assessment; an evaluation of what we could have done had we started once more is also included in this chapter. We end this chapter by investigating the future work, i.e., what ideas have we come up with that may be used for later projects.

8.1 Design and Implementation

Designing network monitoring tasks for TelegraphCQ is a challenging task. As the DSMS is integrated into PostgreSQL, one could assume that most of the features and functionality is implemented in TelegraphCQ as well. But, since TelegraphCQ is part of an ongoing research project, the documentation and supported functionality is limited, or implemented just to investigate some special features. The challenge is therefore to find out what is implemented or not, and possibly interpret the error messages that appear if the tasks are not possible to implement. The TelegraphCQ mailing list has been helpful and the researchers have answered all of our posted questions as quickly as they could.

Many of the main requirements, as described in Section 2.3, are supported in TelegraphCQ. These are, amongst others, *projections*, *selections*, *joins*, and *aggrega-*

tions. We describe some of our experiences with these requirements in the following.

The *projections* are known to choose certain attributes in the tuples. As this feature is used in most queries, TelegraphCQ exhibits good support for projections, and we have not experienced any problems with regard to implementing these. Still, only attributes from the streams can be projected, i.e., we can still not use sub-queries in the `SELECT`-clause, for example. Even though, sub-queries are supported to some extent, as we see below.

Some of the *selections* cause more unexpected results. As mentioned in Chapter 4 and Chapter 5, we experience that there is a maximum of predicates handled by TelegraphCQ when both predicates are parts of SteMs, but we have experienced that this maximum tends to vary between Linux distributions. This limitation is the main cause for the unprecise predicates we use in Task 3. On the other hand, we know from Chapter 4 that the grouped filters are responsible for selections when the predicates have specified values. These selections have worked satisfyingly.

Joining is considered a large overhead in traditional DBMSs, as the results have to be correct, thus complex joins on large windows are heavy and complex operations. For example, the *n-pass algorithms* described in Chapter 3 show this. Plagemann et al. [PGB+04] observe that the join is generally considerably faster than other operators, but that it only performs equi-joins correctly. This is probably caused by the hash indexes in the SteMs. Thus, as we note in Chapter 4, there are ways to go around this problem, by using dummy variables forcing a match on all tuples. We do not investigate this possibility in this thesis since this solution may be considerably inefficient as all tuples are selected. As both relations and streams use hash indexes, Task 2 has covered some of the joining capabilities in TelegraphCQ, and the accuracy is accounted for.

We have investigated some aggregation functions. Task 1 shows e.g. that the *average*, *sum* and *counting* aggregations are supported and accurate when they are used in the projection field. Though, as mentioned above, using count in the `ORDER BY` field, as tested in Task 12, caused the postmaster to stop.

The sub-queries are partly supported in TelegraphCQ, and we see that the `WITH`-clause has its limitations. For instance, Task 3, as we have tested, did not report any tuples, even when it was supposed to. This points out that we need some kind of shedding information about dropped tuples inside the back end. The creation of intermediate tuples increases the load on the eddy. This means that instead of not displaying any tuples, it would have been helpful to have a stream of dropped tuples to show which internal streams dropped most tuples. This would also be help tuning the queries with regard to factors like window size and type.

An implication of sub-queries is that the DSMS manages to handle several concurrent queries. TelegraphCQ uses CACQ to handle this, and we have experienced that TelegraphCQ manages several concurrent queries, since all our experiments were based on running five monitoring queries and the query in the sub-

task concurrently. The third preliminary task partly show this as well, by comparing the result from queries with and without the introspective queries running in the background. Though, nested aggregations are not supported. If we try to include `COUNT(SUM(*))` in a projection, TelegraphCQ states that "`Aggregate function calls may not be nested`".

TelegraphCQ supports a windowing technique that covers the three most common schemes, as introduced in Chapter 3. Due to the design of the tasks, we mostly use the sliding window technique, though Task 6 uses the jumping windows, but this task is not covered thoroughly in this thesis.

The adaptive query processing is an important part of TelegraphCQ [MSHR02, CCD+03, Des04, AH00]. In Task 2, we probably observe this adaptivity, as the harmonic waves converge to a value which is manageable for TelegraphCQ.

With regard to multiplexing and demultiplexing; TelegraphCQ supports demultiplexing by using the `GROUP BY` aggregation. We have experienced that this aggregation works well, as long as the `wtime(*)` function is called in the projection field. If this function is not called in the projection field, we sometimes get an error message saying that the eddy does not know how to handle this query. The multiplexing, or union, is not supported in TelegraphCQ, as shown in Appendix B.3.

The additional requirements, specified for the network monitoring domain, could have been helpful to solve some of our tasks. As mentioned in Task 10, the support for a DSTREAM, which only displays tuples that leave the window, would have been useful in tasks using timeouts for successful queries, for instance. Thus, TelegraphCQ has some limitations with respect to providing protocol states in the query language.

Finally, we also have to use the `text` operator to define the option fields in the stream definition of IP and TCP, as discussed in Chapter 5, instead of letting TelegraphCQ dynamically vary these field's lengths as well as the number of fields.

Above, we show how the requirements described in Section 2.3 is supported in TelegraphCQ. We conclude that TelegraphCQ supports many of the requirements, and is useful for several network monitoring tasks. Still, it does not support some of the requirements that are particularly interesting for network monitoring tasks, such as support for different number of optional header fields.

8.2 Performance Evaluation

The performance evaluation shows how TelegraphCQ behaves in a network monitoring scenario. We create an environment that sends data packets at well defined parameters, such as header size and protocols. We use network load as a factor for varying the input. Based on these characteristics, we send the data stream to TelegraphCQ and measure the relative throughput and results.

We use three metrics to evaluate the DSMS; relative throughput, accuracy, and system consumption. We do not focus on the latter, but rather investigate the two first more deeply. The system consumption is only superficially discussed in Task 2 as part of explaining TelegraphCQ's behavior.

The relative throughput metric shows that TelegraphCQ only manages to handle low traffic load, i.e., 2 to 2.5 Mbits/s depending on the task, before it starts to drop packets. As we have written earlier, these results depend on the qualities of the data stream it receives as well. Though, there are possibilities of adjusting the results by inserting the statistics from the relative throughput into the stream results, as proposed in [RH06] and mentioned in the discussion of Task 1. But this is not assumed to be a plausible solution for all tasks. We show tasks where the adjustment is not possible, for instance, Task 3 and tasks equivalent to it.

TelegraphCQ's accuracy is shown to be satisfying. This may explain the low relative throughput as well; once a packet has entered the eddy, it is sent to the proper modules and possibly not dropped internally. We can not confirm the last statement, since TelegraphCQ does not provide any information about load shedding when the tuples are inside the back end. With regard to accuracy, we sum up the performance evaluation chapter by claiming that tasks that need fine granularity packet information may be too inaccurate to give proper results. Sometimes one would have use for certain packets, for example the ACK packets in the 3-way handshake connection establishment.

With respect to the consumption metric, we do not investigate it too deeply. Still, this is an important aspect in further analysis of TelegraphCQ and how it affects, or is affected by, the system. We need the system monitors mostly to use other information, like the number of packets flowing through the NIC, as described in Task 2, to show the accuracy. As mentioned in Chapter 6, we reduce the total number of possibly disturbing parameters during the run of the experiments. Still, a further study on these parameters may help explaining more about TelegraphCQ.

8.3 Contributions

At the start of this project, finding other articles about both network monitoring and DSMSs was challenging, and we did not manage to find any workbenches for network monitoring and DSMSs particularly. Thus, we decided to build an automatic test environment from scratch. This was also motivated by wanting to fully control the system. The development of `fyaf` was particularly interesting and important. A consequence of this programming is that we have created a system that is capable of sending a well defined data stream to a DSMS through a socket. This probably makes it easier for other projects to verify our results and work on new tasks. At the time of writing, `fyaf` is used as part of two student projects evaluating TelegraphCQ in the "Advanced topics in distributed systems" course at the University of Oslo [inf].

The system we have created supports factors like, *network load*, *packet size*, *number of connections*, *number of runs*, and *length of the runs*. We have verified that the system we have written actually sends data at the expected network loads. We have also tested `fyaf` and verified that its relative throughput is satisfyingly high.

The thesis has also proposed some features which are needed to perform some network monitoring tasks, as mentioned in the previous section.

8.4 Critical Assessment

This section describes what we would have done differently, had we started once more.

Network monitoring is a field that is too large to be covered in one master's thesis. Thus, at the final stages of the thesis we see that there is much more to cover, as well as only a small part the application domain would have been sufficient to focus on. As an example, solving a task concerning SYN flood attack identification would probably be enough for a master's thesis. Designing the queries is challenging, and knowing that the solution designed is the possibly best, is not easy to verify. The design of the queries has been an iterative process, where we had to try several solutions.

However, these are considerations that are taken at the end of the project, where the overview is considerable compared to when the project started. Though, we have managed to do much work, argue for the choices we have made, and we have reached our goals.

The result directories contain much data, because of the introspective queries, the TelegraphCQ log file, and the other monitors that write to file. For example, the directory containing the results from `task_2.1` is 2.6 Gigabytes. It contains eight different network loads rates with five runs each. These are enormous amounts of data, which need considerable time to be analyzed deeply. Hence, we only show some of the possibilities this result set gives. This is also an argument for a performance evaluation of even fewer tasks.

The traffic generator was somewhat cumbersome to use. We had to write scripts that managed to change the configuration files properly. We also had to approach the usage of several connections in an ad-hoc way by creating several instances of the client-server pair explicitly. Due to the time constraints, we did not create our own traffic generator.

TelegraphCQ version 2.0 was released in July 2005. The older version had other window semantics and did not support sub-queries. We had come up with solutions that solved the sub-querying by sending packets back and forth between client instances of TelegraphCQ, but chose to abandon this technique when the new version arrived. As the older version, TelegraphCQ 2.0 tends to stop the without giving any meaningful explanation. This have led to challenges in finding out what queries

the DSMS manages to handle. We have not found the general explanation for TelegraphCQ's sudden stopping.

8.5 Future Work

At the end of this project, as we learn more and get more insight into the problem, we see that there is much more to do.

Due to the traffic generator environment we have created, there are a number of further tasks and metrics that could use this as a base, saving the time of writing such an environment from scratch. Following, is a number of problems that can be solved:

- Introduce factors like header size to measure how much network load TelegraphCQ manages to handle. In our experiments we used a TCP segment size of 576 bytes. This may be varied.
- Investigate only one task, and discuss optimization, window size, and other important factors.
- Our tasks have performed passive network measurement. Try to use TelegraphCQ to analyze results from active measurements, like recursive packet trains (RPTs) of ICMP packets to identify a link's bottleneck [HLM+04].
- Dissect the back end to investigate how packets are scheduled through the different modules. TelegraphCQ consists of hundreds of thousands of code lines and is thus extremely complex.
- Implement B-trees in the SteMs to let more joins than the equi-join work correctly. There is also a possibility of implementing DSTREAMS and other missing functionality.
- Continue investigating the analytical modeling approach that was mentioned while explaining the harmonic reduction observed in Task 2. The window sizes can possibly also be described analytically and derivations may can show the optimal window sizes of several queries, for example.

As our experiments are on-line and the scenarios use real protocols, it would have been interesting to see if there was a possibility of reducing the number of header fields in the `streams.iptcp` definition. This would probably increase the rate TelegraphCQ could handle, but reduce the general applicability of the stream.

Due to the experiment setup we have implemented for investigating the traffic, further analysis of TelegraphCQ queries is simple, as new queries, connection numbers, packet sizes, and number of runs easily can be added to the directories and `sscript.pl`. This is easy, and it saves time for further analysis of tasks, and

opens up for usage of the experiment setup for student projects in advanced data communication courses. There is also a possibility of arranging smaller research projects on some of the topics in shorter master assignments.

Bibliography

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. *Department of Computer Science, Stanford University*, 2004.

[ACC+03] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management, August 2003.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000.

[AW04] Arvind Arasu and Jennifer Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Rec.*, 33(3):6-11, 2004.

[Bab02] Shivnath Babu. Stream query repository: Network traffic management, 2002.
`http://www-db.stanford.edu/stream/sqr/netmon.html`.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 1-16, New York, NY, USA, 2002. ACM Press.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Mike Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD Conf.*, Baltimore, MD, June 2005.

[BSW01] S. Babu, L. Subramanian, and J. Widom. A data stream management system for network traffic management. In *Proceedings of the 2001 Workshop on Network-Related Data Management (NRDM 2001)*, May 2001.

[BW04] Shivnath Babu and Jennifer Widom. Streamon: an adaptive engine for stream query processing. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, page 931-932, New York, NY, USA, 2004. ACM Press.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Frederick Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. *Proceedings of the 2003 CIDR Conference*, 2003.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. page 379-390, 2000.

[CF03] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data, August 2003.

[CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: a language for extracting signatures from data streams. In *Knowledge Discovery and Data Mining*, page 9-17, 2000.

[CFSD90] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP), RFC 1157 (Historic), May 1990. `http://www.ietf.org/rfc/rfc1157.txt`.

[CGJ+02] Chuck Cranor, Y. Gao, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spataschek. Gigascope: High performance network monitoring with an sql interface. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data / Principles of Database Systems*, June 2002.

[CHB+01] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, page 114-130, London, UK, 2001. Springer-Verlag.

[CJ03] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR), RFC 3626 (Experimental), October 2003. `http://www.ietf.org/rfc/rfc3626.txt`.

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 647-651, New York, NY, USA, 2003. ACM Press.

[CK03] Chee-Yee Chong and S. P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247-1256, 2003.
`http://ieeexplore.ieee.org/xpls/abs_all.jsp?-arnumber=1219475`.

[CM99] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations, RFC 2501 (Informational), January 1999.
`http://www.ietf.org/rfc/rfc2501.txt`.

[CVW] Owen Cooper, Ojan Vafai, and Lauren Wilkinson. Recursive query processing in telegraphcq.

[dec] Wikipedia-lookup for declarative programming, 18th of april, 2006.
`http://en.wikipedia.org/wiki/Declarative_programming`.

[Des04] Amol Deshpande. An initial study of overheads of eddies. *ACM SIGMOND*, June 2004.

[Dyn] Telegraphcq dynamic catalog.
`http://telegraph.cs.berkeley.edu/v2.0/-DynCatalog.html`.

[edd] Wikipedia lookup for eddy, 28th of january, 2006.
`http://en.wikipedia.org/wiki/Eddy`.

[EM99] Andrew Eisenberg and Jim Melton. Sql: 1999, formerly known as sql3. *SIGMOD Rec.*, 28(1):131-138, 1999.

[Eur] Institut eurécom.
`http://www.eurecom.fr`.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, page 13-24, New York, NY, USA, 2001. ACM Press.

[GM04] Johannes Gehrke and Samuel Madden. Query processing in sensor networks, 2004.

[GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems, The Complete Book*. Prentice Hall, 2002.

[GNOS05] Eli Gjørven, Tommy André Nyquist, Johannes Oudenstad, and Jarle Søberg. Using a dsms for performing analysis of routing protocols in manets. 2005.

[GP] Vera Goebel and Thomas Plagemann. Data stream management systems (dsms) - applications, concepts, and systems.

[GR02] M. Grossglauser and J. Rexford. Passive traffic measurement for ip operations, 2002.

[GÖ03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5-14, 2003.

[HBP+05] Alefiya Hussain, Genevieve Bartlett, Yuri Pryadkin, John Heidemann, Christos Papadopoulos, and Joseph Bannister. Experiences with a continuous network tracing infrastructure. In *MineNet '05: Proceeding of the 2005 ACM SIGCOMM workshop on Mining network data*, page 185-190, New York, NY, USA, 2005. ACM Press.

[Her06] Kjetil Helge Hernes. Design, implementation, and evaluation of networks monitoring tasks with the stream data stream management system, May 2006.

[HLM+04] Ningning Hu, Li (Erran) Li, Zhuoqing Morley Mao, Peter Steenkiste, and Jia Wang. Locating internet bottlenecks: algorithms, measurements, and implications. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, page 41-54, New York, NY, USA, 2004. ACM Press.

[ian] Internet assigned numbers authority. `http://www.iana.org/`.

[inf] Inf5090 - advanced topics in distributed systems. `http://www.uio.no/studier/emner/matnat/ifi/INF5090/`.

[Jai91] Rai Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[JC99] Theodore Johnson and Damianos Chatziantoniou. Extending complex ad-hoc olap. In *CIKM '99: Proceedings of the eighth international conference on Information and knowledge management*, page 170-179, New York, NY, USA, 1999. ACM Press.

[JHP05] Sandra K. Johnson, Gerrit Huizenga, and Badari Pulavarty. *Performance Tuning for Linux Servers*. IBM Press, Pearson plc, May 2005.

[JMSS05] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava. Streams, security and scalability. In *Proceeding of the 19th Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec 2005)*, August 2005.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel R. Madden, Frederick Reiss, and Mehul A. Shah. Telegraphcq: An architectual status report. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2003.

[KS05] Jürgen Krämer and Bernhard Seeger. A temporal foundation for continuous queries over data streams. page 70-82, 2005.

[Loa] Load shedding in telegraphcq. `http://telegraph.cs.berkeley.edu/v2.0/Triage.html`.

[MDK+00] G. Montenegro, S. Dawkins, M. Kojo, V. Magret, and N. Vaidya. Long Thin Networks, RFC 2757 (Informational), January 2000. `http://www.ietf.org/rfc/rfc2757.txt`.

[MF02] S Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, February 2002.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122-173, 2005.

[MLD02] Paul E. McKenney, Dan Y. Lee, and Barbara A. Denny. *Traffic generator release notes*, 2002.

[Mou] Kyriakos Mouratidis. Data stream processing: An overview of recent research.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive contiuous queries over streams. *ACM SIGMOND*, June 2002.

[pca] pcap - packet capture library. `http://www.tcpdump.org/pcap3_man.html`.

[PGB+04] Thomas Plagemann, Vera Goebel, Andreas Bergamini, Giacomo Tolu, Guillaume Urvoy-Keller, and Ernst W. Biersack. Using data stream management systems for traffic analysis - a case study. April 2004.

[Plu82] D.C. Plummer. Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware, RFC 826 (Standard), November 1982. `http://www.ietf.org/rfc/rfc826.txt`.

[Pos] *PostgreSQL 7.3.2 Users Guide*.

[Pos80] J. Postel. User Datagram Protocol, RFC 768 (Standard), August 1980.
`http://www.ietf.org/rfc/rfc768.txt`.

[Pos81a] J. Postel. Internet Protocol, RFC 791 (Standard), September 1981. Updated by RFC 1349,
`http://www.ietf.org/rfc/rfc791.txt`.

[Pos81b] J. Postel. Transmission Control Protocol, RFC 793 (Standard), September 1981. Updated by RFC 3168,
`http://www.ietf.org/rfc/rfc793.txt`.

[Pos83] J. Postel. TCP maximum segment size and related topics, RFC 879, November 1983.
`http://www.ietf.org/rfc/rfc879.txt`.

[PR85] J. Postel and J.K. Reynolds. File Transfer Protocol, RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773,
`http://www.ietf.org/rfc/rfc959.txt`.

[pro] Wikipedia-lookup for procedural programming, 18th of april, 2006.
`http://en.wikipedia.org/wiki/Procedural_programming`.

[Pun] Punctuation in telegraphcq.
`http://telegraph.cs.berkeley.edu/v2.0/-Punctuation.html`.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. *Report No. UCB/CSD-03-1231*, February 2003.

[Rei] Frederick Reiss. Tcq window semantics.
`http://telegraph.cs.berkeley.edu/v2.0/-Windows.html`.

[RH01] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *The VLDB Journal*, page 381-390, 2001.

[RH04] Frederick Reiss and Joseph M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. Report, February 2004.

[RH06] Frederick Reiss and Joseph M. Hellerstein. Declarative network monitoring with an underprovisioned query processor. In *The 22nd International Conference on Data Engineering (to appear)*, April 2006.

[RL93] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR, RFC 1518 (Proposed Standard), September 1993. `http://www.ietf.org/rfc/rfc1518.txt`.

[SBG+05] Matti Siekkinen, Ernst W. Biersack, Vera Goebel, Thomas Plagemann, and G. Urvoy-Keller. Intrabase: Integrated traffic analysis based on a database management system. May 2005.

[SG90] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262-266, 1990.

[Sie06] Matti Siekkinen. Measuring the internet: State of the art and challenges, February 2006.

[SLP05] A. Spognardi, A. Lucarelli, and R. D Pietro. A methology for p2p file-sharing traffic detection. In *Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems (HOT-P2P)*, July 2005.

[SMFH01] Mehul A. Shah, Sam Madden, Michael J. Franklin, and Joseph M. Hellerstein. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record*, 30(4):103-114, 2001.

[Tela] The telegraph project. `http://telegraph.cs.berkeley.edu/`.

[Telb] The telegraphcq homepage. `http://telegraph.cs.berkeley.edu/v2.0`, `http://telegraph.cs.berkeley.edu/telegraphcq/-v0.2/`.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, page 321-330, New York, NY, USA, 1992. ACM Press.

[Wra] Wrappers in telegraphcq. `http://telegraph.cs.berkeley.edu/v2.0/-wrapperwriter.html`, `http://telegraph.cs.berkeley.edu/-v2.0/Data_Acquisition.html`.

[YG03] Y. Yao and J. E. Gehrke. Query processing for sensor networks. In *Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003)*, January 2003.

[ZS02] Y. Zhu and D Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, August 2002.

[ZSC+03] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects, 2003.

Appendix A

Acronyms

Throughout the thesis, we have used a large number of acronyms, and when introduced once, there might be challenging to remember all of them. The acronyms are ordered alphabetically.

ADC	Analogue to Digital Converter
CPU	Central Processing Unit
CQ	Continuous Query
CSV	Comma Separated Value
DoS	Denial-of-Service
DBMS	Data Base Management System
DSMS	Data Stream Management System
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
MANET	Mobile Ad-hoc NETwork
MSS	Maximum Segment Size
NIC	Network Interface Card
P2P	Peer-to-Peer
QRQ	Query Result Queue
RAM	Random Access Memory
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TCP	Transfer Control Protocol
WCH	Wrapper Clearing House
SteM	State Module
UDP	User Datagram Protocol

Appendix B

Tests

This chapter is dedicated to the smaller tests that are referred to during the thesis' discussions. Each test is explained in its presentation.

B.1 The "»" *Cidr* Operator

B.1.1 Introduction and Design

This test will see if the "»" cidr operator works properly. We create a table containing some subnets, and send a stream of cidr addresses to a query that joins the stream and the table using "»".

We create a table having 24 b subnet indicators; `192.168.1/24` to `192.168.4-/24`. The task is to join with a table and then join with a stream. Both the table and the stream contains addresses that fall under one of the subnets. The table, which is used in the table join resembles `streams.iptcp` defined in Figure 5.1. The prefix table and insertions are presented in Figure B.1. The queries are presented in Figure B.2. We observe the differences.

B.1.2 Experiment

We send the datafile to the DSMS for the streams. Firstly, we just project all attributes from the stream to see if TelegraphCQ manages to receive the stream. Then we join the prefixes and the stream. Finally, we test joins with the table and the prefixes.

B.1.3 Results and Conclusion

The first test showed that TelegraphCQ managed to obtain data from the stream. The second test did not give any results, and the third test gave correct results.

Figure B.1 The table defining the prefixes.

```
DROP TABLE department;

CREATE TABLE department (
        name text,
        prefix cidr
);

INSERT INTO department VALUES (
        'A',
        '192.168.1/24'          10
);

INSERT INTO department VALUES (
        'B',
        '192.168.2/24'
);

INSERT INTO department VALUES (
        'C',
        '192.168.3/24'          20
);

INSERT INTO department VALUES (
        'D',
        '192.168.4/24'
);

\i shift_table.sql
\i shift_stream.sql
```

Figure B.2 The table and stream queries for "»" testing.

```
-- name: shiftTable.sql

SELECT
        department.name, count(*)
FROM
        tables.iptcp,
        department
WHERE
        department.prefix >> tables.iptcp.sourceIP
GROUP BY                                    10
        department.name;



-- name: shiftStream.sql

SELECT
        department.name, count(*)
FROM
        streams.iptcp
        [RANGE BY '1 second' SLIDE BY '1 second'],
        department
WHERE                                       20
        department.prefix >> streams.iptcp.sourceIP
GROUP BY
        department.name;
```

We conclude that the "»" *cidr* operator does not work in TelegraphCQ version 2.0.

B.2 Join on Non-Equalities within Streams

B.2.1 Introduction and Design

Sometimes it is important to project tuples that are not part of a join. In several tests, we have experienced that TelegraphCQ does not print any results when we test this. Thus, we have created a test case that verifies that the results are not correct, i.e., something should have been output.

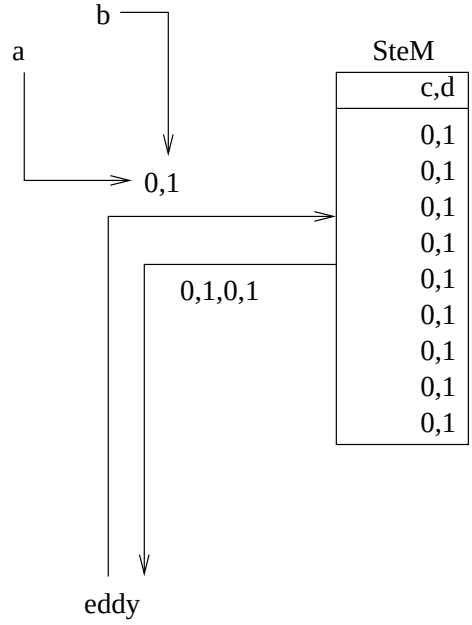
We define a stream consiting of two distinct tuples having attributes `a`, `b`, `c`, and `d`:

```
a|b|c|d
-------
1,1,0,1
0,1,1,1
```

Then we create a query having a `WITH` clause:

```
WITH
        streams.a_b_test AS
        (
        SELECT a, b, wtime(*)
        FROM streams.a_b_c_d_test
        [RANGE BY '10 seconds' SLIDE BY '1 second']
        WHERE a = 0 AND b = 1
        GROUP BY a, b
        )

        streams.c_d_test AS
        (
        SELECT c, d, wtime(*)
        FROM streams.a_b_c_d_test
        [RANGE BY '10 seconds' SLIDE BY '1 second']
        WHERE c = 0 AND d = 1
        GROUP BY c, d
        )

(SELECT
        *
FROM
        streams.a_b_test AS ab
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
```

Figure B.3 Expected behavior for non-equality join.




```
        streams.c_d_test AS cd
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        ab.b <> cd.c);
```

As we see, 100 % of the stream is supposed to be displayed; the ones where `ab.b` is not equal to `cd.c`. Figure B.3 shows the expected behavior. The eddy sends the `<a,b>` tuple to the SteM corresponding to the `<c,d>` tuples. The intermediate tuple is then returned to the eddy.

B.2.2 Results and Conclusion

Following is a sample of the output:

```
|    | 2006-03-24 13:52:14 |    |    | 2006-03-24 13:52:13
|    | 2006-03-24 13:52:14 |    |    | 2006-03-24 13:52:14
|    | 2006-03-24 13:52:15 |    |    | 2006-03-24 13:52:14
|    | 2006-03-24 13:52:15 |    |    | 2006-03-24 13:52:15
|    | 2006-03-24 13:52:16 |    |    | 2006-03-24 13:52:15
|    | 2006-03-24 13:52:16 |    |    | 2006-03-24 13:52:16
```

As we see, the results are empty. We see two outputs per timestamp. This shows the empty punctuation tuple. Based on this observation we are forced to conclude that TelegraohCQ does not support non-equality joins on streams.

B.3 UNION, EXCEPT, and INTERSECT

On several occations, we have needed functionality for set operators. Examples are merging of streams, or removing certain tuples. We created three simple tests using `streams.a_b_c_d_test`. The results are displayed like this:

```
master_db=# (SELECT a FROM streams.a_b_c_d_test)
master_db-# UNION
master_db-# (SELECT b FROM streams.a_b_c_d_test);
ATTENTION: Running in CQ mode but not sending this
query as a CQ query
 a
---
(0 rows)

master_db=# (SELECT a FROM streams.a_b_c_d_test)
master_db-# EXCEPT
master_db-# (SELECT b FROM streams.a_b_c_d_test);
ATTENTION: Running in CQ mode but not sending this
query as a CQ query
 a
---
(0 rows)
master_db=# (SELECT a FROM streams.a_b_c_d_test)
master_db-# INTERSECT
master_db-# (SELECT b FROM streams.a_b_c_d_test);
ATTENTION: Running in CQ mode but not sending this
query as a CQ query
 a
---
(0 rows)
```

When we enter a set operator in TelegraphCQ, it displays that it does not send the qurey as a continuous query. We conclude that TelegraphCQ does not support the set operators `UNION`, `EXCEPT`, and `INTERSECT`.

B.4 TCP SYN Timeout Interval

B.4.1 Introduction and Design

The number of retries of TCP SYN packets is set to five per default, but may change from different operating systems or settings. With five retransmits, the SYN

are sent for approximately three minutes. We see if this is the case in our test environment.

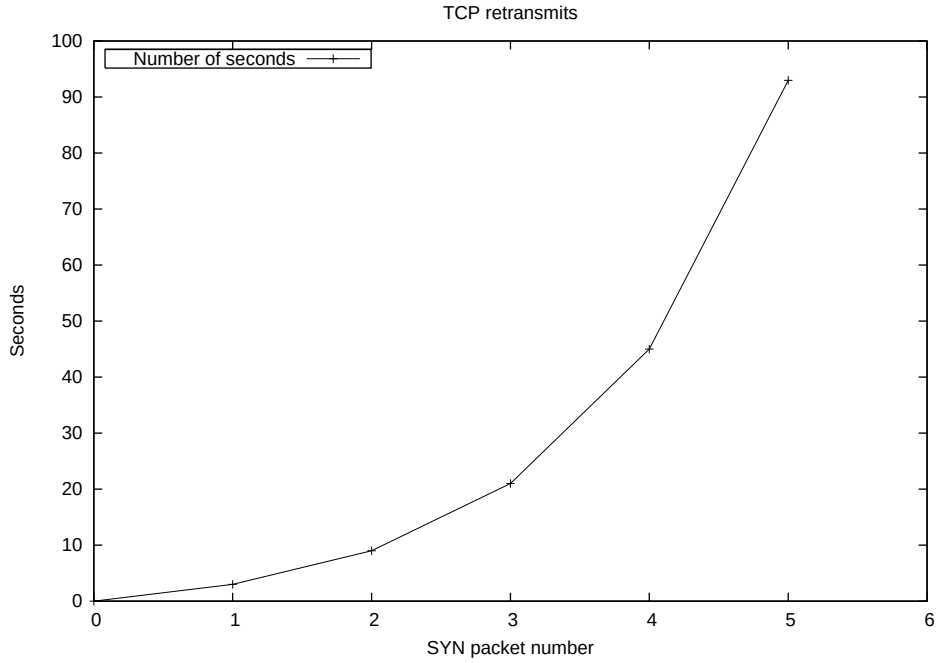
B.4.2 Experiment

We run two instances, one that tries to telnet to an address and a port that does not exist using `time`, and one that listens to the network card using `tcpdump`. `tcpdump` puts an epoch timestamp on all the packets. We use this information to calculate the difference between the first and the following packets. We run the experiment 10 times and calculate the average.

B.4.3 Results and Conclusion

The telnet command timed out after an average of 188.973 seconds. This was after six packets were sent; one initial packet and five retires. Figure B.4 shows that the result corresponds with the assumption. There are five retries, and we see that the time interval doubles for each retry.

Figure B.4 The result from testing number of retries.



Appendix C

TelegraphCQ Files

C.1 Description of the Introspective Streams

The introspective streams are new features in TelegraphCQ version 2.0. Following is a description of the streams, as defined in [Dyn]. The descriptions are located in Tables C.1, C.2, and C.3.

Column	Type	Description
`tcqtime`	`timestamp without time zone`	Timestamp of the event
`qrynum`	`integer`	Sequentially increasing query number
`qid`	`integer`	Result queue associated with this query
`kind`	`character(1)`	Nature of the event: '`E`' - Entry '`X`' - Exit
`qrystr`	`character varying(1000)`	Query string

Table C.1: Definision of `tcq_queries`.

Column	Type	Description
tcqtime	timestamp without time zone	Timestamp of the event
opnum	integer	Sequentially increasing operator number
opid	integer	Current operator identifier associated with this operator
numqrs	integer	Not commented
kind	character(1)	Nature of the event: 'E' - Enter operator (brand new operator created) 'A' - Add query to operator (operator folded with existing operator) 'R' - Remove query from operator 'X' - Exit operator
qid	integer	Queue that corresponds to the query that caused this operator event
opstr	character varying(1000)	Exact textual description of the Node data structure corresponding to this operator
opkind	character(1)	Nature of this operator 'F' - FSteM 'G' - GSFilter 'M' - ScanModule
opdesc	character varying(100)	Readable textual description of the expression associated with this operator

Table C.2: Definision of tcq_operators.

Column	Type	Description
`tcqtime`	`timestamp without time zone`	Timestamp of the event
`opid`	`integer`	Operator associated with this queue
`qkind`	`character(1)`	Nature of this queue '`I`' - Input Queue '`O`' - Output Queue
`kind`	`character(1)`	Nature of the event '`E`' - Successful enqueue '`F`' - Failed enqueue '`D`' - Successful dequeue '`N`' - NULL dequeue

Table C.3: Definision of `tcq_queues`.

Appendix D

Tables for Preliminary Task 3

Relative throughput reported by system.

Network load (Mbits/s)	task_101	task_101.1	task_101.2	task_101.3	task_101.4	task_101.5
1	99.981	99.987	99.988	99.902	99.980	99.983
5	55.018	95.536	98.606	86.637	99.996	99.996
10	29.843	43.510	45.031	46.385	72.788	79.639
20	3.131	4.042	4.128	7.352	8.820	10.868
25	0.565	1.561	0.534	2.848	2.115	5.709
30	0.249	0.393	0.404	0.493	0.699	0.610
40	0.169	0.339	0.342	0.298	0.361	0.409

Relative throughput reported by TelegraphCQ.

Network load (Mbits/s)	task_101	task_101.1	task_101.2	task_101.3	task_101.4	task_101.5
1	100.000	100.000	100.000	100.000	100.000	100.000
5	91.851	97.998	98.610	100.000	100.000	100.000
10	35.885	43.543	45.031	62.049	79.824	84.288
20	3.309	4.042	4.128	8.307	8.928	10.936
25	0.684	1.561	0.534	3.014	2.116	5.719
30	0.402	0.405	0.405	0.560	0.713	0.610
40	0.327	0.350	0.369	0.337	0.361	0.409

The two results compared.

Network load (Mbits/s)	task_101	task_101.1	task_101.2	task_101.3	task_101.4	task_101.5
1	-0.019	-0.013	-0.012	-0.098	-0.020	-0.017
5	-36.833	-2.462	-0.004	-13.363	-0.004	-0.004
10	-6.042	-0.033	-0.000	-15.664	-7.036	-4.649
20	-0.178	-0.000	0.000	-0.955	-0.108	-0.068
25	-0.119	-0.000	0.000	-0.166	-0.001	-0.010
30	-0.153	-0.012	-0.001	-0.067	-0.014	0.000
40	-0.158	-0.011	-0.027	-0.039	0.000	0.000

Table D.1: Overview of the relative throughput in Preliminary Task 3.

Runs with reported errors

Network load (Mbits/s)	task_101	task_101.1	task_101.2	task_101.3	task_101.4	task_101.5
1	run3	OK	OK	OK	OK	OK
5	OK	OK	OK	OK	OK	OK
10	OK	OK	run6	OK	OK	OK
20	OK	OK	OK	OK	OK	OK
25	OK	OK	OK	OK	OK	OK
30	OK	OK	OK	OK	OK	OK
40	OK	OK	OK	OK	OK	OK

Empty runs due to `fyaf` failure.

Network load (Mbits/s)	task_101	task_101.1	task_101.2	task_101.3	task_101.4	task_101.5
1	OK	run0,run2	OK	OK	OK	OK
5	run0,run6	run0,run1	run5,run7	OK	OK	OK
10	run3,run6,run8	run2	OK	OK	OK	OK
20	OK	OK	OK	OK	OK	OK
25	OK	OK	OK	OK	OK	OK
30	OK	OK	OK	OK	OK	OK
40	OK	OK	OK	OK	OK	OK

Table D.2: Overview of error prone or empty runs in Preliminary Task 3.

Appendix E

Task 1 Supplements

E.1 Accuracy

We send a stream of 10 packets each second having a `totalLength` of 628 is sent to the DSMS. The results show 10 packets and 50,240 bits per second, which corresponds to an accurate answer. The result files are located in `/dmms-lab65/-tcq_scripts/experiment_client/accuracy/Task1/` on the DVD-ROM.

E.2 Queries, Tables, and Plots

Figure E.1 Setup for `streams.task1_1`.

```
DROP STREAM streams.task1_1;

CREATE STREAM streams.task1_1 (
        totalNum int,
        totalLength int,
        tcqtime timestamp TIMESTAMPCOLUMN
) TYPE UNARCHIVED ON OVERLOAD KEEP COUNTS;

ALTER STREAM streams.task1_1 ADD WRAPPER csvwrapper;
```

Figure E.2 Setup for `streams.task1_2`.

```
DROP STREAM streams.task1_2;

CREATE STREAM streams.task1_2 (
        totalNum int,
        totalLength int,
        tcqtime timestamp TIMESTAMPCOLUMN
) TYPE UNARCHIVED ON OVERLOAD KEEP COUNTS;

ALTER STREAM streams.task1_2 ADD WRAPPER csvwrapper;
```

Runs with reported errors.

Network load (Mbits/s)	`task_1.1`	`task_1.2`
1	OK	OK
2	OK	OK
2.5	OK	OK
5	OK	OK
7.5	OK	OK
10	OK	OK

Empty runs due to `fyaf` failure.

Network load (Mbits/s)	`task_1.1`	`task_1.2`
1	OK	OK
2	OK	OK
2.5	run2	OK
5	OK	OK
7.5	OK	OK
10	OK	OK

Table E.1: Overview of error prone or empty runs in `task_1.1` and `task_1.2`.

Percent kept from `task_1.1` and `task_1.2`.

Network load (Mbits/s)	`task_1.1`	`task_1.2`
1	100.000	100.000
2	100.000	100.000
2.5	100.000	100.000
5	99.522	97.003
7.5	69.141	66.909
10	45.545	44.138

Table E.2: Overview of `task_1.1` and `task_1.2`.

Calculated relative throughput based on TelegraphCQ's output.

Network load (Mbits/s)	`task_1.1`
1	97.146
2	100.860
2.5	99.580
5	98.668
7.5	68.596
10	44.782

Reported relative throughput based on TelegraphCQ's shedding mechanism.

Network load (Mbits/s)	`task_1.1`
1	100.000
2	100.000
2.5	100.000
5	99.522
7.5	69.141
10	45.545

Table E.3: Calculated throughput vs. reported relative throughput from TelegraphCQ in Task 1.

Network load (Mbits/s)	`task_1.5`
1	100.000
2	100.000
2.5	100.000
5	91.085
7.5	61.477
10	40.011

Table E.4: Relative throughput for packets for `task_1.5`.

Figure E.3 Network load reported by `task_1.1`.

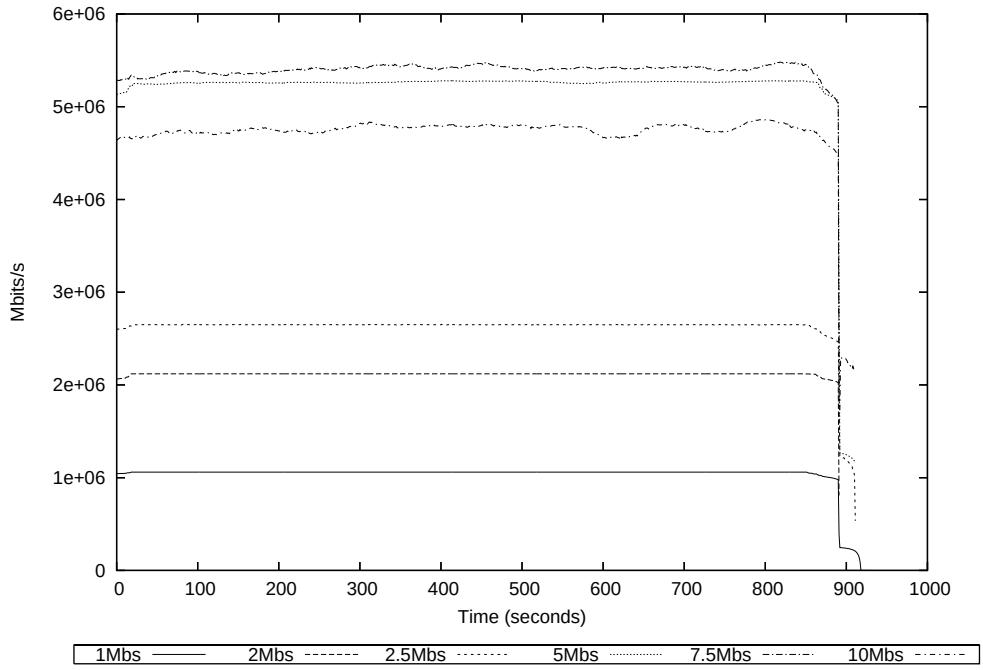


Figure E.4 Network load reported by `task_1.2`.

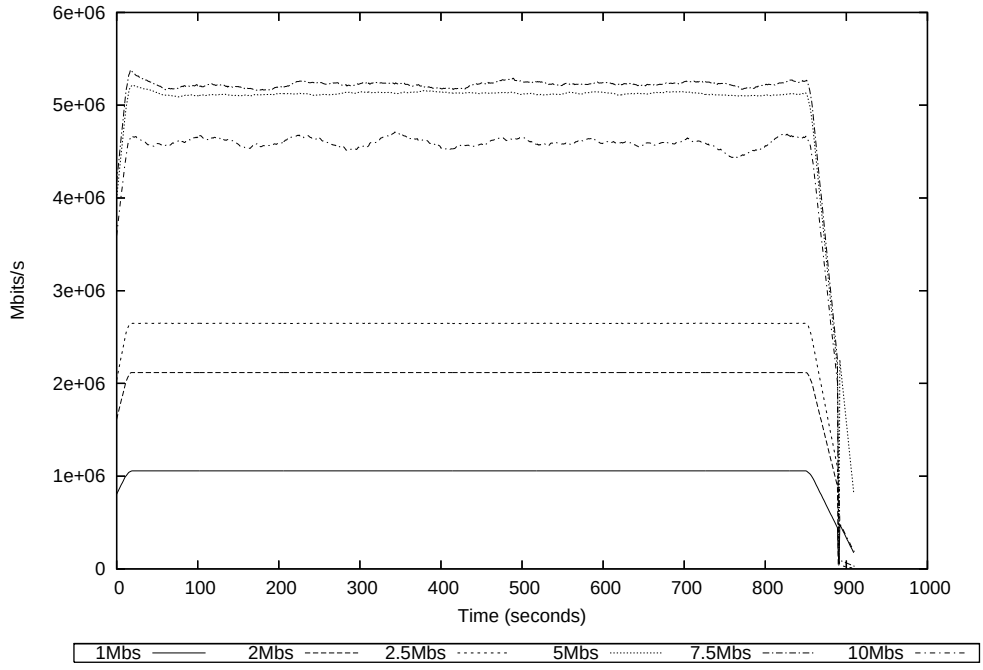


Figure E.5 Number of packets reported by `task_1.1`.

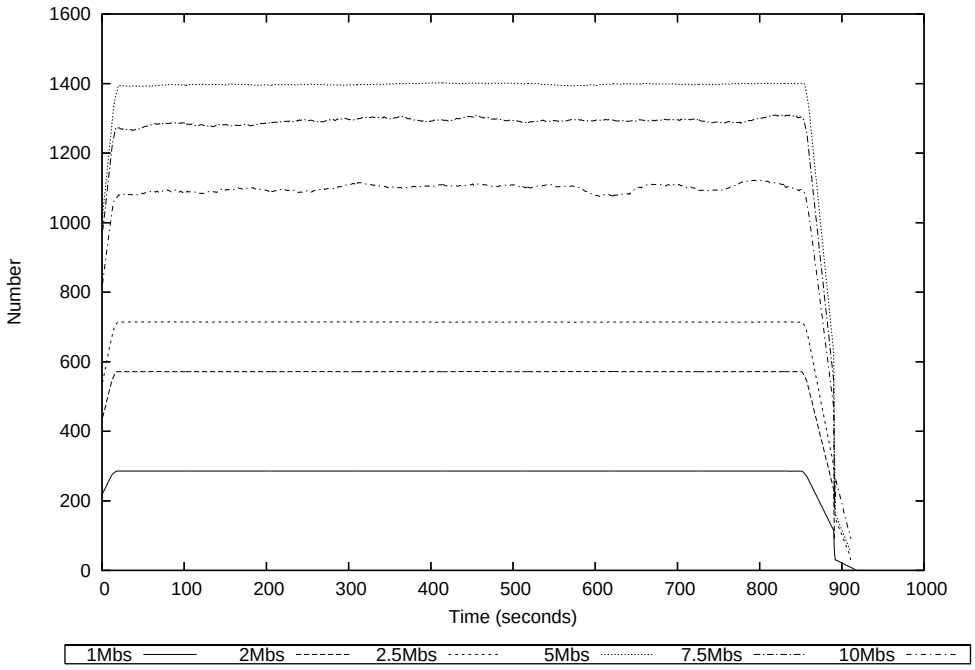


Figure E.6 Number of packets reported by `task_1.2`.

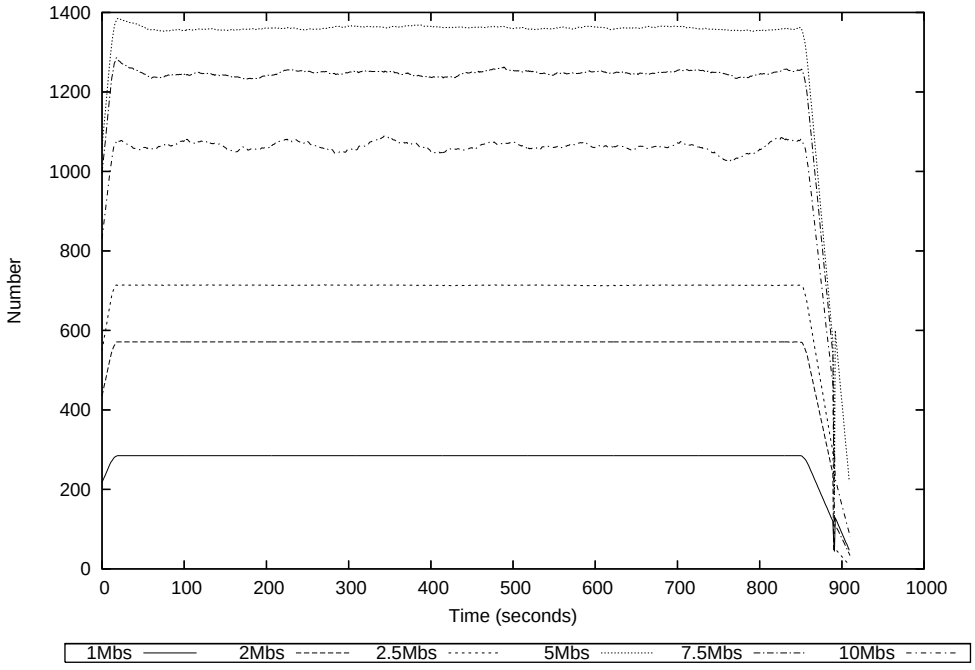
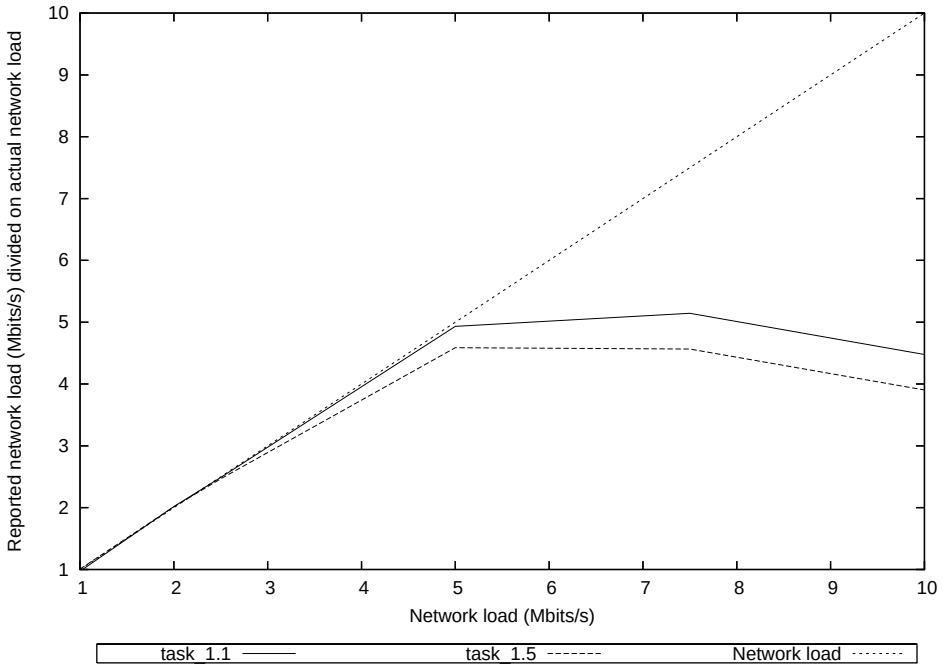


Figure E.7 Network load compared between `task_1.1` and `task_1.5`.



Appendix F

Task 2 Supplements

F.1 Accuracy

We use send 10 packets each second heading for port 60,010. The result shows approximatly 2,980 accesses per five minutes. $\frac{2,980}{5}/60 = 9.93$, which we find acceptable. The result files are located in `/dmms-lab65/tcq_scripts/-experiment_client/accuracy/Task2/` on the DVD-ROM.

F.2 Tables and Plots

Runs with reported errors

Network load (Mbits/s)	`task_2.1`	`task_2.2`	`task_2.3`
1	OK	OK	OK
1.2	OK	not tested	not tested
2.5	run3	OK	OK
3	run1,run2,run4	OK	run3
3.75	OK	run2	OK
5	OK	OK	OK
7.5	OK	OK	OK
10	OK	run4	OK

Empty runs due to `fyaf` failure.

Network load (Mbits/s)	`task_2.1`	`task_2.2`	`task_2.3`
1	OK	OK	OK
1.2	OK	not tested	not tested
2.5	run0,run1,run2,run4	OK	OK
3	run0,run3	run0,run1,run2,run3,run4	run0,run1,run2,run4
3.75	run1,run2	OK	run0
5	OK	OK	run0
7.5	run1	OK	OK
10	OK	OK	run3

Table F.1: Overview of error prone or empty runs in the sub-tasks for Task 2.

Network load (Mbits/s)	`task_2.1`	`task_2.2`	`task_2.3`
1	47615.387	47959.457	48003.771
1.2	58567.402	not tested	not tested
2.5	N/A	121510.026	121944.330
3	N/A	N/A	N/A
3.75	69502.282	126919.736	127772.023
5	65003.841	121629.646	122665.291
7.5	60956.190	109127.518	107484.710
10	55262.128	97321.995	96916.151

Table F.2: Number of packets destined `dmms-lab60`.

Network load (Mbits/s)	`task_2.1`	`task_2.2`	`task_2.3`
1	100.000	100.000	100.000
1.2	100.000	not tested	not tested
2.5	N/A	99.991	99.997
3	N/A	N/A	N/A
3.75	45.363	73.110	73.450
5	34.975	56.612	56.251
7.5	23.228	37.962	37.737
10	15.397	25.316	25.476

Table F.3: Relative throughput for the sub-tasks in Task 2.

Figure F.1 Number of packets destined for `dmms-lab60` in `task_2.1`.

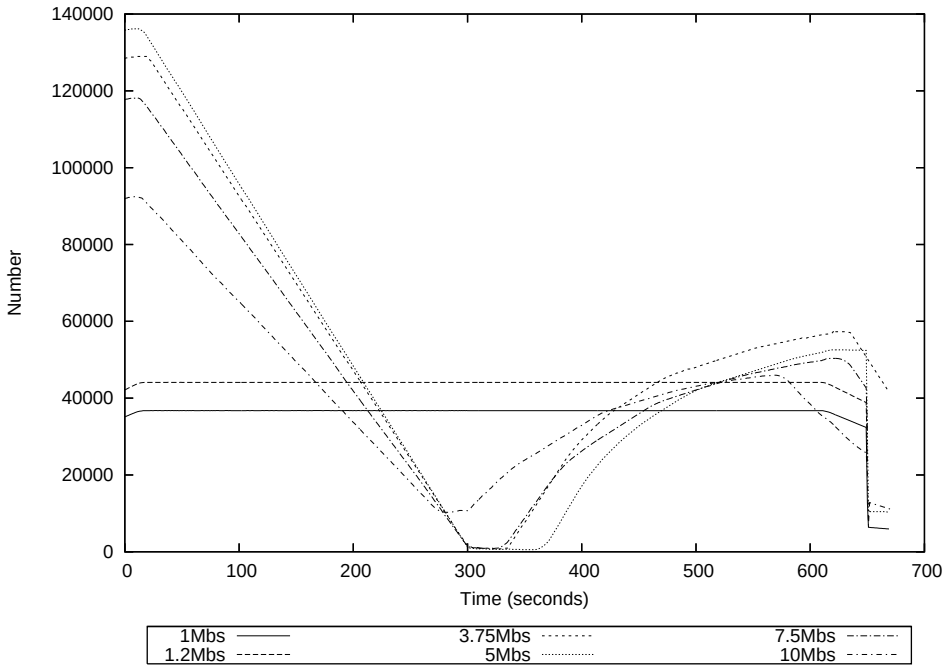
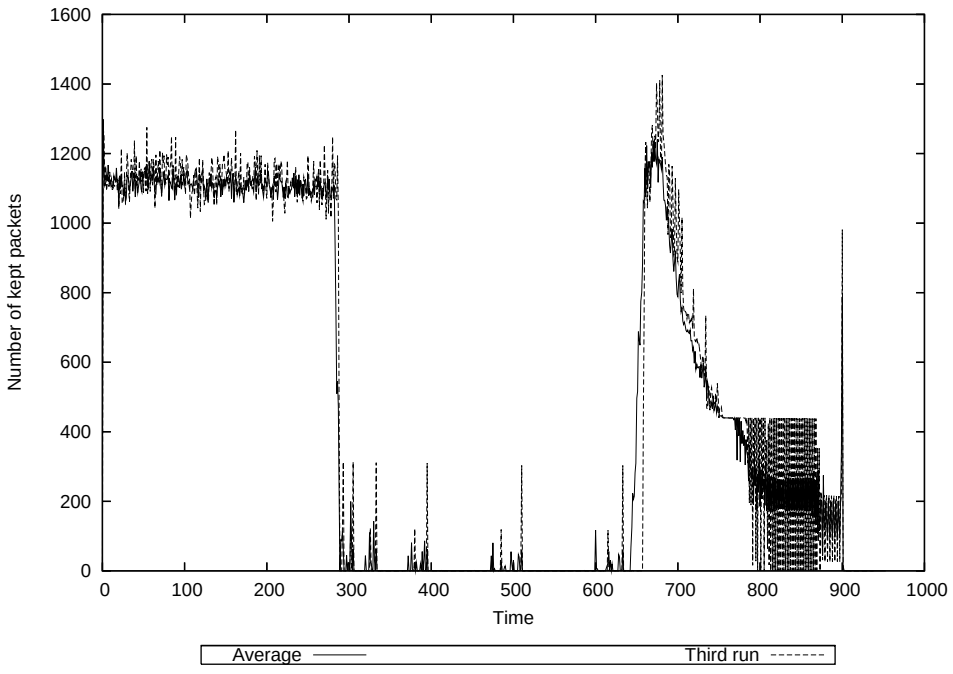


Figure F.2 A visualisation of differences between a selected run versus the calculated average.



Appendix G

Task 3 Supplements

G.1 Accuracy

We only investigate the connection establishment query. We establish three connections, which is correctly reported in the result files located in `/dmms-lab65/-tcq_scripts/experiment_client/accuracy/Task3/` on the DVD-ROM. We send one tuple each second.

G.2 Queries, Results, Tables, and Plots

Runs with reported errors

Network load (Mbits/s)	`task_3.1`	`task_3.2`	`task_3.3`
0.01	OK	not tested	not tested
0.1	run4	OK	OK
1	OK	run1,run2,run4	OK

Empty runs due to `fyaf` failure.

Network load (Mbits/s)	`task_3.1`	`task_3.2`	`task_3.3`
0.01	OK	not tested	not tested
0.1	OK	OK	OK
1	OK	run0,run3	OK

Table G.1: Overview of error prone or empty runs in base tasks for Task 3.

Following are the results from projecting all tuples from the run at 0.1 Mbits/s from `streams.syn` and `streams.synack`, respectively (we shorten the number of punctuation tuples and insert newlines to save space):

```
,,,,,,2006-04-05 19:30:38.873598
```

```
,,,,,,2006-04-05 19:30:39.87438
,,,,,,2006-04-05 19:30:40.877163
,,,,,,2006-04-05 19:30:41.87794
129.240.67.60/32,129.240.67.65/32,8173,60010,3480332800,
0,2006-04-05 19:30:42.001008
129.240.67.60/32,129.240.67.65/32,8174,60011,3476085725,
0,2006-04-05 19:30:42.00137
129.240.67.60/32,129.240.67.65/32,8175,60012,3477045056,
0,2006-04-05 19:30:42.00178
129.240.67.60/32,129.240.67.65/32,8176,60013,3481404964,
0,2006-04-05 19:30:42.002821
129.240.67.60/32,129.240.67.65/32,8177,60014,3476906664,
0,2006-04-05 19:30:42.003098
129.240.67.60/32,129.240.67.65/32,8178,60015,3490220034,
0,2006-04-05 19:30:42.003588
,,,,,,2006-04-05 19:30:42.878722
,,,,,,2006-04-05 19:30:43.879494
,,,,,,2006-04-05 19:30:43.879494
,,,,,,2006-04-05 19:30:44.880296
,,,,,,2006-04-05 19:30:45.881058
,,,,,,2006-04-05 19:30:46.881843
,,,,,,2006-04-05 19:30:47.882622
,,,,,,2006-04-05 19:30:48.882684
,,,,,,2006-04-05 19:30:49.8842
,,,,,,2006-04-05 19:30:50.884979
,,,,,,2006-04-05 19:30:51.885762
129.240.67.60/32,129.240.67.65/32,8179,60016,3485373534,
0,2006-04-05 19:30:52.000395
129.240.67.60/32,129.240.67.65/32,8180,60017,3494579658,
0,2006-04-05 19:30:52.002022
129.240.67.60/32,129.240.67.65/32,8181,60018,3495694414,
0,2006-04-05 19:30:52.00258
129.240.67.60/32,129.240.67.65/32,8182,60019,3493832619,
0,2006-04-05 19:30:52.003007
,,,,,,2006-04-05 19:30:52.886546
,,,,,,2006-04-05 19:30:53.887318

,,,,,,2006-04-05 19:30:39.87438
,,,,,,2006-04-05 19:30:40.877163
,,,,,,2006-04-05 19:30:41.87794
129.240.67.65/32,129.240.67.60/32,60010,8173,2229253433,
3480332801,2006-04-05 19:30:42.001219
129.240.67.65/32,129.240.67.60/32,60011,8174,2227781047,
3476085726,2006-04-05 19:30:42.00151
129.240.67.65/32,129.240.67.60/32,60012,8175,2224530097,
3477045057,2006-04-05 19:30:42.002263
129.240.67.65/32,129.240.67.60/32,60013,8176,2229551607,
3481404965,2006-04-05 19:30:42.002959
129.240.67.65/32,129.240.67.60/32,60014,8177,2230514405,
3476906665,2006-04-05 19:30:42.003241
129.240.67.65/32,129.240.67.60/32,60015,8178,2225483288,
3490220035,2006-04-05 19:30:42.003721
,,,,,,2006-04-05 19:30:42.878722
,,,,,,2006-04-05 19:30:43.879494
```

```
,,,,,,2006-04-05 19:30:43.879494
,,,,,,2006-04-05 19:30:44.880296
,,,,,,2006-04-05 19:30:45.881058
,,,,,,2006-04-05 19:30:46.881843
,,,,,,2006-04-05 19:30:47.882622
,,,,,,2006-04-05 19:30:48.882684
,,,,,,2006-04-05 19:30:49.8842
,,,,,,2006-04-05 19:30:50.884979
,,,,,,2006-04-05 19:30:51.885762
129.240.67.65/32,129.240.67.60/32,60016,8179,2240979357,
3485373535,2006-04-05 19:30:52.001822
129.240.67.65/32,129.240.67.60/32,60017,8180,2242031990,
3494579659,2006-04-05 19:30:52.002166
129.240.67.65/32,129.240.67.60/32,60018,8181,2232836200,
3495694415,2006-04-05 19:30:52.002722
129.240.67.65/32,129.240.67.60/32,60019,8182,2239457486,
3493832620,2006-04-05 19:30:52.003143
,,,,,,2006-04-05 19:30:52.886546
,,,,,,2006-04-05 19:30:53.887318
,,,,,,2006-04-05 19:30:54.888099
```

The SQL query, which uses tables instead of streams to verify the Task 3 is located at `/dmms-lab65/tcq_scripts/master_tasks/task_3.3_rev/task3-_3_query_only_conn.sql`. The table setup is located in the `/dmms-lab65/-tcq_scripts/master_tasks/` directory. The results are shown in the accuracy directory for Task 3 but is shown here as well. Since we read from a file, the `num_wraps` attribute shows the result from a `COUNT(*)` and not discussed due to time limitations:

```
master_db=# select * from tables.conn;
     sourceip     |      destip      | sourceport | destport | num_wraps
------------------+------------------+------------+----------+-----------
 129.240.67.60/32 | 129.240.67.65/32 |      13482 |    60016 |     83815
 129.240.67.60/32 | 129.240.67.65/32 |      13483 |    60010 |     84547
 129.240.67.60/32 | 129.240.67.65/32 |      13484 |    60011 |     84034
 129.240.67.60/32 | 129.240.67.65/32 |      13485 |    60012 |     84064
 129.240.67.60/32 | 129.240.67.65/32 |      13486 |    60013 |     84064
 129.240.67.60/32 | 129.240.67.65/32 |      13487 |    60014 |     84133
 129.240.67.60/32 | 129.240.67.65/32 |      13488 |    60015 |     83995
 129.240.67.60/32 | 129.240.67.65/32 |      13489 |    60019 |     83928
 129.240.67.60/32 | 129.240.67.65/32 |      13490 |    60017 |     83758
 129.240.67.60/32 | 129.240.67.65/32 |      13491 |    60018 |     83565
success=0, cqcancel=0
```

Figure G.1 The query that shows the conn queries. Note that the figures follow eachother as part of the queries.

```
WITH
        streams.syn AS
        (
        SELECT
                sourceIP, destIP, sourcePort, destPort,
                seqNum, ackNum, tcqtime
        FROM
                streams.iptcp
        WHERE
                SYN = '1' AND ACK = '0'                          10
        )

        streams.synack AS
        (
        SELECT
                sourceIP, destIP, sourcePort, destPort,
                seqNum, ackNum, tcqtime
        FROM
                streams.iptcp
        WHERE                                                    20
                SYN = '1' AND ACK = '1'
        )

        streams.conn AS
        (
        SELECT
                syn.sourceIP, syn.destIP,
                syn.sourcePort, syn.destPort, wtime(*)
        FROM
                streams.syn AS syn                               30
                [RANGE BY '180 seconds' SLIDE BY '1 seconds'],
                streams.synack AS synack
                [RANGE BY '180 seconds' SLIDE BY '1 seconds'],
                streams.iptcp AS ack
                [RANGE BY '180 seconds' SLIDE BY '1 seconds']
        WHERE
                syn.sourceIP = synack.destIP
                AND syn.destIP = synack.sourceIP
                AND syn.sourcePort = synack.destPort
                AND syn.destPort = synack.sourcePort             40
                AND ack.SYN = '0' AND ack.ACK = '1'
                AND synack.seqNum + 1 = ack.ackNum
        GROUP BY
                syn.sourceIP, syn.destIP, syn.sourcePort,
                syn.destPort
        )
```

Figure G.2 The query that shows the i/rpayload queries.

```
streams.ipayload AS
(
SELECT
        s.sourceIP, s.sourcePort, s.destIP, s.destPort,
        sum(s.totalLength) - (sum(s.ipHeaderLength)*4)
        - (sum(s.tcpHeaderLength)*4), wtime(*)
FROM
        streams.conn AS conn
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.iptcp AS s                                          10
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        conn.sourceIP = s.sourceIP
        AND conn.destIP = s.destIP
        AND conn.sourcePort = s.sourcePort
        AND conn.destPort = s.destPort
GROUP BY
        s.sourceIP, s.sourcePort, s.destIP, s.destPort
)
                                                                    20
streams.rpayload AS
(
SELECT
        s.destIP, s.destPort, s.sourceIP, s.sourcePort,
        sum(s.totalLength) - (sum(s.ipHeaderLength)*4)
        - (sum(s.tcpHeaderLength)*4), wtime(*)
FROM
        streams.conn AS conn
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.iptcp AS s                                          30
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE
        conn.sourceIP = s.destIP
        AND conn.destIP = s.sourceIP
        AND conn.sourcePort = s.destPort
        AND conn.destPort = s.sourcePort
GROUP BY
        s.sourceIP, s.sourcePort, s.destIP, s.destPort
)
                                                                    40
```

Figure G.3 This is the final query that joins the ipayload and rpayload streams.

```
(SELECT
        i.sourceIP, i.sourcePort, i.destIP, i.destPort,
        sum(i.totalBytes) + sum(r.totalBytes) as totalBytes,
        wtime(*)
FROM
        streams.ipayload AS i
        [RANGE BY '10 seconds' SLIDE BY '1 second'],
        streams.rpayload AS r
        [RANGE BY '10 seconds' SLIDE BY '1 second']
WHERE                                                          10
        r.sourceIP = i.destIP
        AND r.destIP = i.sourceIP
        AND r.sourcePort = i.destPort
        AND r.destPort = i.sourcePort
GROUP BY
        i.sourceIP, i.sourcePort, i.destIP, i.destPort);
```

Appendix H

Tests for the Other Tasks

The tasks and results are mostly located in the DVD-ROM. Each of the following task's descriptions describes where.

H.1 Task 4

We use a table with two distinct tuples; one heading for port 80 and one for port 21. We send 2 packets each second. The stream results seem to stabilize on 90 for port 21 and 100 for 80. The expected result is 100 for both. We try to switch the two `WITH` clause sub-queries, but the results are the same. Though, it seems to be a constant, but unexpected inaccuracy in the merging technique. The result files are located in `/dmms-lab65/tcq_scripts/experiment_client-/accuracy_tests/Task_4/` in the DVD-ROM.

H.2 Task 6

The problem with this task is that it does not give any output; the postmaster process stops. One concern is that the queries within the `WITH`-clause does not send information to eachother. We test this by a simple query located in `/dmms-lab65/-tcq_scripts/master_tasks/task_6_rev/task_6_verify.sql` in the DVD-ROM. The query sends data from `streams.iptcp` to another, which again sends to the final query. This works at least, something which indicates that the reason for the stopping has something to do with the aggreagtions. The result files are located in `/dmms-lab65/tcq_scripts/experiment_client/-accuracy_tests/Task_6/` in the DVD-ROM. Still, we leave this task to future work.

Figure H.1 The simplified query showing sums of several windows compressed.

```
WITH
        streams.minutes
        AS
        (
        SELECT
                sourceIP, sourcePort, destIP, destPort,
                sum(totalLength−ipHeaderLength
                −tcpHeaderLength)
                AS dataSum, wtime(*)
        FROM                                                    10
                streams.iptcp
                [RANGE BY '1 seconds' SLIDE BY '1 second']
        GROUP BY
                sourceIP, sourcePort, destIP, destPort
        )

        streams.hours
        AS
        (
        SELECT                                                  20
                sourceIP, sourcePort, destIP, destPort,
                sum(dataSum)
                AS dataSum, wtime(*)
        FROM
                streams.minutes
                [RANGE BY '1 seconds' SLIDE BY '1 seconds']
        GROUP BY
                sourceIP, sourcePort, destIP, destPort
        )
                                                                30
(SELECT
        sourceIP, sourcePort, destIP, destPort,
        sum(dataSum)
        AS dataSum, wtime(*)
FROM
        streams.hours
        [RANGE BY '1 seconds' SLIDE BY '1 seconds']
GROUP BY
        sourceIP, sourcePort, destIP, destPort);
                                                                40
```

H.3 Task 8

The query is located in `/dmms-lab65/tcq_scripts/master_tasks/-task_8_rev/task_8.sql`, and the `/dmms-lab65/tcq_scripts/experiment_client/task_8_rev/task8.res`. The file shows the result from a Web browsing session manually configured. Sum the number of each window to verify that the result is correct.

H.4 Task 9

The query is located in `/dmms-lab65/tcq_scripts/master_tasks/-task_9_rev/task9_join.sql`. The simple results are located in `/dmms--lab65/tcq_scripts/experiment_client/task_9_rev/task_9_-adding.res`

H.5 Task 11

The query is located in `/dmms-lab65/tcq_scripts/master_tasks/-task_11_rev/`. The test dumpfile is located there as well. It contains three test-case tuples that resemble the IP and TCP packet headers. The result file is located in `/dmms-lab65/tcq_scripts/experiment_client/task_11-_rev/task11.res` and shows a set containing many of the same tuple; the correct one.

H.6 Task 12

The query is located in `/dmms-lab65/tcq_scripts/master_tasks/-task_12_rev/`, and the result file is located in `/dmms-lab65/tcq_scripts-/experiment_client/task_12_rev/task12.res`. See that the result corresponds to how it is described in Section 5.3.9.

Appendix I

Implementation Selections

I.1 `fyaf` Source Code Selection

This section only contains a selection of the `fyaf` source code. The complete source code is located in the DVD-ROM as described in Appendix K.

I.1.1 `fyaf_IP4.h`

```
#if !defined _OPTIONS_
#define _OPTIONS_
typedef struct {
    u_int data;
} OPTIONS;
#endif

#if !defined _IP4_
#define _IP4_

typedef struct {
    u_short div1; /* version
                     headerLength
                     tos
                  */
    u_short totalLength;
    u_short id;

    u_short div2; /* flags
                     fragOffset
                  */
    u_short div3; /* ttl
```

```
                        protocol
                    */
    u_short headChksum;
    u_int sourceIP;
    u_int destIP;
} IP4;

u_int printIP4(int , IP4 * , int * , u_char * );
#endif
```

I.1.2 fyaf_IP4.c

```
#include "fyaf7.h"
#include "fyaf_IP4.h"
#include <netinet/in.h>

#define IP4HLEN 5

#ifndef FYAFBYTES
#define FYAFBYTES 4
#endif

#define IP4HSIZE IP4HLEN*FYAFBYTES

u_int printIP4(int IP4set, IP4 * ip4, int *
               add_to_addr, u_char * outp)
{
  int i = 0;
  int cnt = 0;
  int headerLength;

  OPTIONS * options;

  int
    div1,
    div2,
    div3;

  div1 = ntohs(ip4->div1);
  div2 = ntohs(ip4->div2);
  div3 = ntohs(ip4->div3);

  if(IP4set)
```

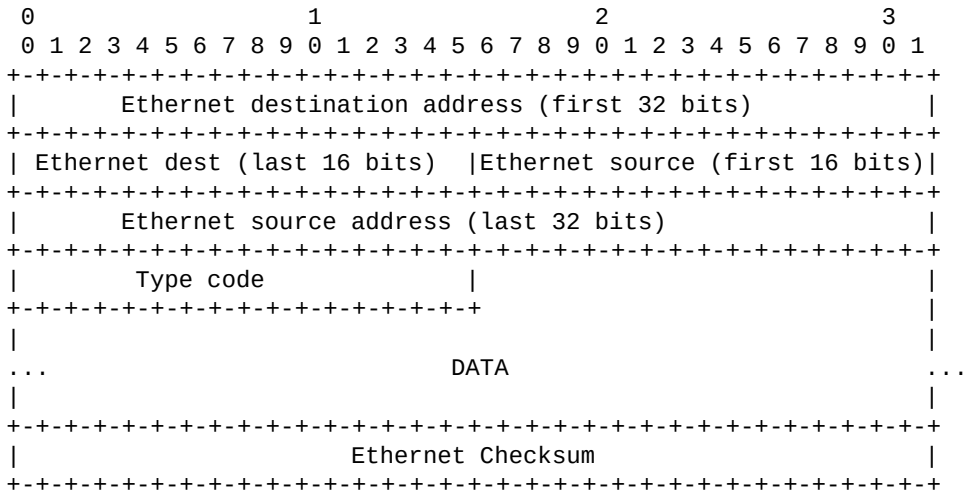
```
{
  cnt += sprintf(outp,
                 "%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,"
                 "%d.%d.%d.%d,%d.%d.%d.%d,",
                 div1>>12,       /* version */
                 (div1>>8)&15,   /* headerLength */
                 div1&127,       /* tos */
                 ntohs(ip4->totalLength),
                 ntohs(ip4->id),                                  40
                 div2>>13,       /* flags */
                 div2&8191,      /* fragOffset */
                 div3>>8,        /* ttl */
                 div3&127,       /* protocol */
                 ntohs(ip4->headChksum),
                 (unsigned int)((ip4->sourceIP & 0x000000FF)),
                 (unsigned int)((ip4->sourceIP & 0x0000FF00) >> 8),
                 (unsigned int)((ip4->sourceIP & 0x00FF0000) >> 16),
                 (unsigned int)((ip4->sourceIP & 0xFF000000) >> 24),
                                                                  50
                 (unsigned int)((ip4->destIP & 0x000000FF)),
                 (unsigned int)((ip4->destIP & 0x0000FF00) >> 8),
                 (unsigned int)((ip4->destIP & 0x00FF0000) >> 16),
                 (unsigned int)((ip4->destIP & 0xFF000000) >> 24)
      );

  /*
     Since the DSMSs may have problems with
     changing number of attributes in a stream,
     all options are merged together and written as hex.          60
     If there are no options, a 0 is written.
  */

  outp += cnt;

  headerLength = (div1>>8)&15;

  *add_to_addr = headerLength*FYAFBYTES;

  if(headerLength == IP4HLEN)                                     70
  {
       cnt += sprintf(outp, "0");

       return cnt;
  }
```

```
        options = (OPTIONS *)ip4+IP4HLEN;

        for(i = 0; i < (headerLength−IP4HLEN); i++)
        {                                                          80
            cnt += sprintf(outp,
                           "%08x",
                           ntohl(options−>data));
            options = (OPTIONS *) options+1;
            outp += 8;
        }

        *add_to_addr = headerLength*FYAFBYTES;

        return cnt;                                                90
      }

    *add_to_addr = 0;

    return 0;
}
```

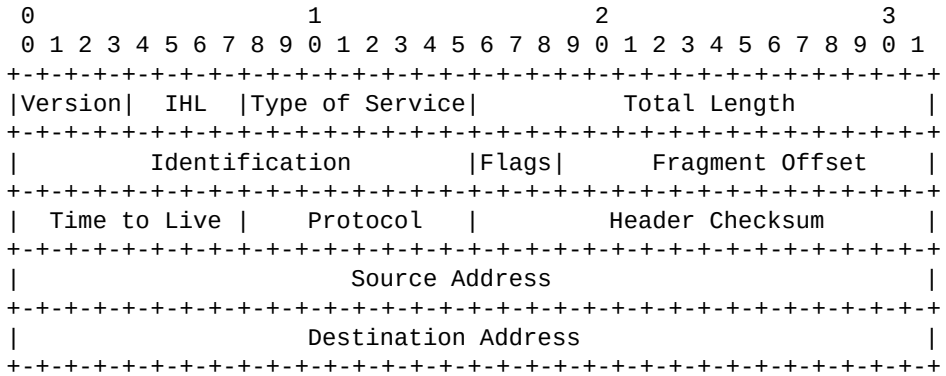
Appendix J

Packet Headers

J.0.3 The Ethernet Header, IEEE 802



J.0.4 The IPv4 Header [PR85]



```
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

J.0.5 The TCP Header [Pos81a]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

J.0.6 The UDP Header [Pos80]

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

Appendix K

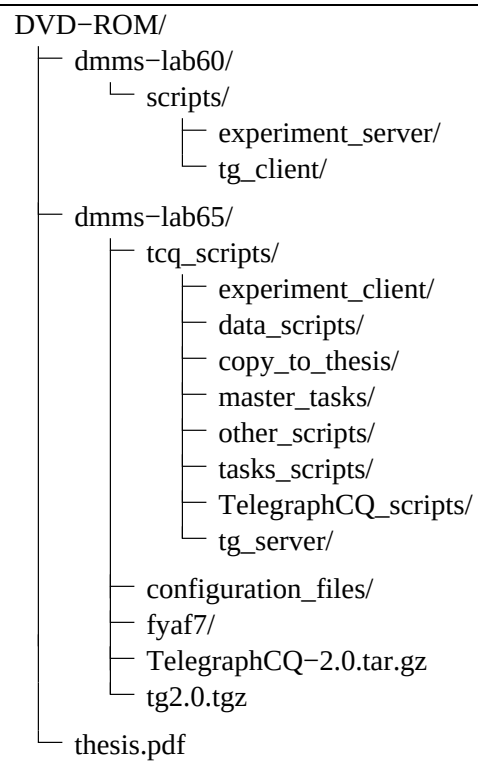
The DVD-ROM

K.1 General Overview

The DVD-ROM contains all the data files and scripts used in this thesis. It also contains a version of this document. Figure K.1 gives an overview of some of the directories in the DVD-ROM. Follwing is a short description of what each of the directories contains (We have written several README files in the directories):

- `dmms-lab60/` contains the client scripts for sending data. The machine also acts as a server at the start of each experiment.
  - `experiment_server/` contains one single script, `experiment_server.pl`, which is accepting instructions from `dmms-lab65` at the beginning of the experiments.
  - `tg_client/` contains the scripts that modifies the TG configuration files.
- `dmms-lab65/` contains a considerable amount of files, since this is the machine where most of the computation is done.
  - `tcq_scripts/` is where the main script directories are located. The directories contain README files that inform about the most important files. Following is a short description.
    - `experiment_client/` contains the results from the performance evaluation and corresponding scripts.
    - `data_scripts/` contains scripts that are used to change the format of the data files, calculate average, and create LaTeX tables.
    - `copy_to_thesis/` contains files that are zipped and copied to the thesis directory.
    - `master_tasks/` contains the design of the tasks.

Figure K.1 An overview of the DVD.



    - `other_scripts/` contains other scripts that are used to help e.g. stopping processes.
    - `tasks_scripts/` contains the scripts that computes the data from each of the performance evaluation tasks.
    - `TelegraphCQ_scripts/` contains scripts that are used to e.g. start and stop TelegraphCQ.
    - `tg_server/` contains the configuration files for the TG server(s).
  - `configuration_files` contains the `postgresql.conf` and `telegraphcqinit.h`.
  - `fyaf7/` contains the complete source code for `fyaf`.

When exploring the DVD, one may see that many of the result files are zipped and tared. The files takes about 29 Gbytes, which means that any further evaluation of the files depend on unpacking the files.

“Bandwidth” on the DVD files means “network load” due to some factor name changes in the late stages of the experiments.

K.2 Re-Testing the Experiments

In case of re-testing the experiments, first of all, use Linux. TelegraphCQ is solely developed in Linux, and can only be used in such enviroments. To make the experiments work, there are several things to do. We try to give a short overview.

To begin with, copy the files in `dmms-lab65/` to the user's root, i.e., `/home-/username/`. `tcq_scripts/` is then located at `/home/username/tcq-_scripts/`. The `dmms-lab60/` files are correspondingly copied to the other machine.

One must set up the experiment in `sscript.pl`. The file contains several setups that show how to run experiments. Only a slight knowledge about Perl is needed to understand the logic. The scripts are statically set up to run against `dmms-lab60` and `dmms-lab65`. Either re-configure the scripts, or the computers used in the experiments. Also make sure that the scripts in the different directories in `tcq_-scripts/` are in the path.

The tasks that are tested, are located in `tcq_scripts/master_tasks/`. Look at the `task_*` directories to see how the files are structured. Each of these directories contains a `rum.pl` script, which are called by `super_script2.pl`.

When set up, simply run `sscript.pl` and hope for the best. When the experiments are finished, they are located in the corresponding `task_*` directory in `tcq_scripts/experiment_client/`. In this directory, the experiments are further located in a directory telling what time the experiments started.

If one wants to continue by hand, one enters the `task_*` directory that contains the new results. Then write `move_to_new_task.pl task_*`. This script copies the files from the dates and prints them as network load directories in the `task_*` directory. Note that if the current network load directory already exists, an error may appear. In such cases, it is sufficient to rename the runs so that the already existing runs are not disturbed. In that case, write `change3.pl` with a start number of the new runs as an argument, and then write `move_to_new_task.pl task_*`. If everyting have worked correctly, the `task_*` directory is now containing directories that indicate the various network loads.

The next thing to do is to run `create_average2.pl` with the `task_*` directory as argument. This script creates a standard set of average numbers dependent on the results. These results are only basic and only contains information about relative throughput, and number of correct runs and simular data.

To run the computation of the results referred to on the thesis, simply unpack the result files by writing `. ./UNPACK`, and write `. ./RUNME` in the `experiment_client/` directory. This script calls simular `RUMNE` scripts in the `task-_*_sum/` directories. The scripts that are custom buildt for each of the tasks are located in the `tcq_scripts/tasks_scripts/` directory.